

Direct Draw

P R O G R A M M I N G

CD-ROM
with editing
tools, games
and source
code

- Learn how to create superfast graphics for killer games and multimedia applications
- Optimized for use with major C/C++ compilers for Windows 95 or Windows NT
- Master the sprite class, tiling class, and object class, and implement them together
- Maximize your programming success by using the right drivers and configuration



B r e t T i m m i n s



DIRECTDRAW PROGRAMMING

Bret Timmins



M&T Books

A Division of MIS:Press, Inc.

A Subsidiary of Henry Holt and Company, Inc.

115 West 18th Street

New York, New York 10011

© 1996 by M&T Books

Printed in the United States of America

All rights reserved. No part of this book may be reproduced or transmitted in any form or by any means, electronic or mechanical, including photocopying, recording, or by any information storage and retrieval system, without prior written permission from the Publisher. Contact the Publisher for information on foreign rights.

Limits of Liability and Disclaimer of Warranty

The Author and Publisher of this book have used their best efforts in preparing the book and the programs contained in it. These efforts include the development, research, and testing of the theories and programs to determine their effectiveness.

The Author and Publisher make no warranty of any kind, expressed or implied, with regard to these programs or the documentation contained in this book. The Author and Publisher shall not be liable in any event for incidental or consequential damages in connection with, or arising out of, the furnishing, performance, or use of these programs.

All products, names and services are trademarks or registered trademarks of their respective companies.

Library of Congress Cataloging-in-Publication Data

Timmins, Bret.

DirectDraw programming / by Bret Timmins.

p. cm.

ISBN 1-55851-460-0

1. Computer graphics. 2. Computer animation. 3. DirectDraw.

4. Computer games. I. Title.

T385.T56 1996

794.8'16765-dc20

9554175

CIP

97 96 95 94 4 3 2 1

Associate Publisher: Paul Farrell

Editor in Chief: Cary Sullivan

Development Editor: Andy Neusner

Copy Editor: Karen Tongish

Technical Editor: David Petchey

Copy Chief: Shari Chappell

C O N T E N T S

Chapter One: An Introduction to DirectDraw 1

Topics Covered in this Book	1
PC Graphics, A Brief History	2
Super VGA	3
Enter Windows	4
DirectDraw Features	4

Chapter Two: How to Get the Most from this Book 7

Installing the Game SDK	7
Finding a DirectDraw Video Driver	9
C/C++ Compilers Supported	10
Using ANSI C	11
The Win32 API	11

Chapter Three: DirectDraw Concepts 13

The DirectDraw COM	14
The DirectDraw Model	15
DirectDraw Communication	15
The GDI and DirectDraw	16

Chapter Four: The DirectDraw Object 17

The DirectDrawCreate Function	17
The SetCooperativeLevel function	18
The DirectDraw GetCaps function	19
The DDCAPS Structure	19
The Connect Program	20

Chapter Five: Creating Simple DirectDraw Surfaces: The Primary Surface 27

The Surface Metaphor	27
Linear Memory Access	28
Surface Properties	29
The Primary Surface	29
Surface Creation	30
The DDSURFACEDESC Structure	31
The SetDisplayMode Function	32
The RestoreDisplayMode Function	32
Creating a Primary Surface Step by Step	33
The DirectDrawSurface Object	33
The Lock Function	34
The Unlock Function	34
The DirectDrawSurface Release Function	35
The PRIMARY.CPP program	35

Chapter Six: Creating DirectDraw Surfaces: The Back Buffer and Page Flipping 45

Page Flipping	45
Page Flipping Under DirectDraw	46
The Flip Mechanism	47
Creating a Back Buffer	48
The GetAttachedSurface function	49
The Flip Function	50
The BACKBUFF.CPP Program	50

Chapter Seven: Creating DirectDraw Surfaces: The Off-Screen Surface 59

Being Off Screen	59
Creating an Off-Screen Surface	60
The System Memory Surface	61
Blting	62

The BltFast function	62
The OFF_SCR.CPP Program	64
Using Off-Screen Surfaces as Buffers	74
The OFFSCREEN1.CPP Program	74

Chapter Eight: Blting: Transparent Blting 87

The Transparent Pixel	87
Using Transparency for Depth Effects	88
The Color Key	88
The DDCOLORKEY Structure	89
The SetColorKey Function	89
The GetColorKey Function	90
The TB_SRC.CPP Program	91
The TB_DEST.CPP Program	103

Chapter Nine: Blting: Blt Effects 117

Blt or BltFast?	117
The Blt Function	118
DDBLT_FX	120
The Color Fill	121
The SHRINK.CPP Program	121

Chapter Ten: Blting: DirectDraw Clippers 133

Clipping	133
Clipping Under DirectDraw	135
The CreateClipper Function	135
Creating Clip Lists	136
The SetHWND Function	136
The SetClipList Function	137
Using the RCNDATA structure	137
A SetClipList Example	138
The SetClipper Function	139
Clipper Limitations	139
The CLIP.CPP Program	140

Chapter Eleven: DirectDraw Color 153

Palette Modes	153
Palettes and DirectDraw	154
The CreatePalette function	155
The SetPalette Function	156
The SetEntries and GetEntries Functions	157
Palette-Independent Color	158
Hicolor RGB Video Modes	158
The GetSurfaceDesc Function	159
The DDPIXELFORMAT Structure	160
The HICOLOR.CPP Program	160

Chapter Twelve: Running Windowed 175

The GDI and the Primary Surface	175
Running in a Normal Window	176
Writing to the Primary Surface Window	176
Using the GDI	178
The GetDC Function	178
The ReleaseDC Function	178
Finding the GDI	178
The FlipToGDISurface Function	179
Palette Control	179
Palette Control and DirectDraw	180
The WINDOW.CPP Program	180

Chapter Thirteen: Debugging DirectDraw Programs 201

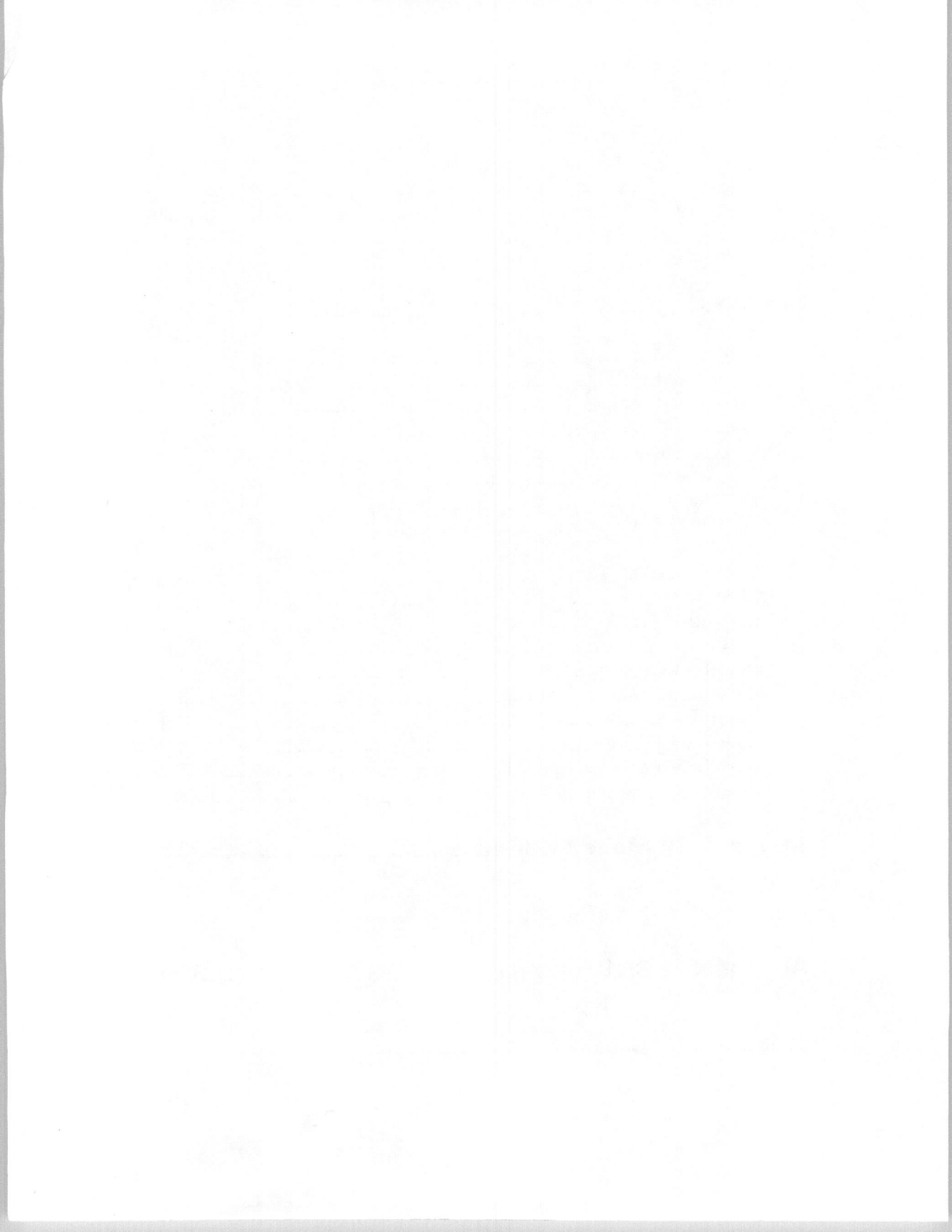
The GDI Debugger	201
A Debugger Compromise	202
Kernel-Level Debuggers	203
The DEBUG.CPP Program	204

Appendix A: DirectDraw Reference 219

DirectDraw APIs	219
DirectDrawCreate	219
DirectDrawEnumerate	220
DirectDraw Member Function Reference	221
AddRef	221
Compact	221
CreateClipper	222
CreatePalette	223
CreateSurface	224
DuplicateSurface	225
EnumDisplayModes	226
EnumSurfaces	228
FlipToGDISurface	230
GetCaps	230
GetDisplayMode	231
GetFourCCCodes	232
GetGDISurface	232
GetMonitorFrequency	233
GetScanLine	234
GetVerticalBlankStatus	234
Initialize	235
QueryInterface	236
Release	236
SetCooperativeLevel	237
SetDisplayMode	238
WaitForVerticalBlank	240
DirectDrawClipper Member Function Reference	241
AddRef	241
GetClipList	241
GetHWND	242
Initialize	243
IsClipListChanged	243
QueryInterface	244

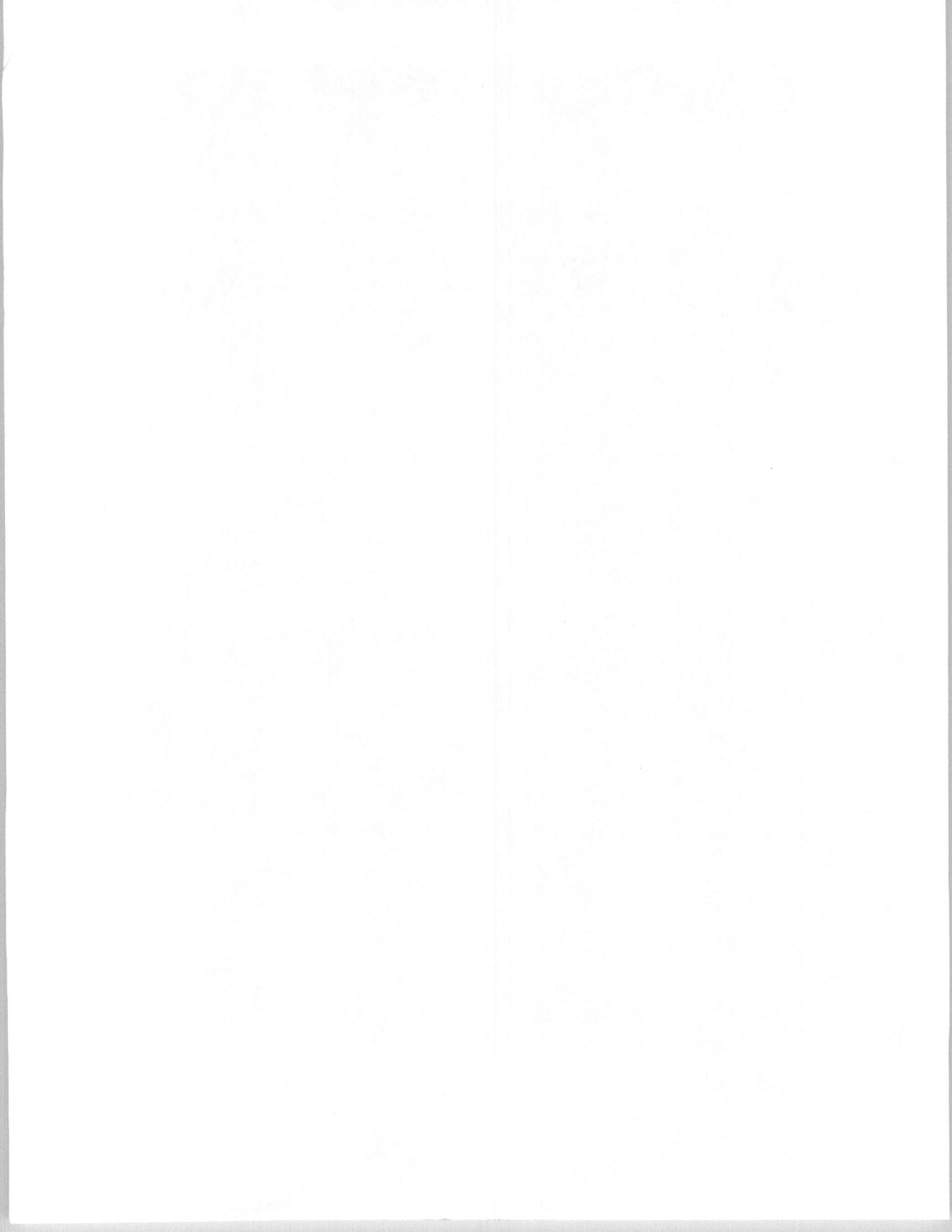
Release	245
SetClipList	245
SetHWND	246
DirectDrawPalette Member Function Reference	247
AddRef	247
GetCaps	247
GetEntries	248
Initialize	249
QueryInterface	250
Release	251
SetEntries	251
DirectDrawSurface Member Function Reference	253
AddAttachedSurface	253
AddOverlayDirtyRect	253
AddRef	254
Blt	255
BltBatch	259
BltFast	261
lpSrcRect	262
DeleteAttachedSurfaces	263
EnumAttachedSurfaces	264
EnumOverlayZOrders	265
Flip	266
GetAttachedSurface	267
GetBltStatus	268
GetCaps	269
GetClipper	269
GetColorKey	270
GetDC	271
GetFlipStatus	272
GetOverlayPosition	273
GetPalette	274
GetPixelFormat	274
GetSurfaceDesc	275
Initialize	276
IsLost	276
Lock	277
QueryInterface	279

Release	280
ReleaseDC	280
Restore	281
SetClipper	282
SetColorKey	282
SetOverlayPosition	283
SetPalette	284
Unlock	285
UpdateOverlay	286
UpdateOverlayDisplay	289
UpdateOverlayZOrder	290
DirectDraw Structures	292
DDCAPS	298
DDCOLORKEY	309
DDMODEDESC	309
DDOVERLAYFX	310
DDPIXELFORMAT	312
DDSCAPS	315
DDSURFACEDESC	316
DirectDraw	319
DirectDrawClipper	320
DirectDrawPalette	320
DirectDrawSurface	321
DirectDraw Return Values	322
DirectDraw Enumeration Call Back Return Values	322
DirectDraw Error Return Values	322
Appendix B: ModeX Surfaces	333
ModeX Restrictions	333
The MODEX.CPP Program	334
Appendix C: Bibliography	349
Index	351



A C K N O W L E D G M E N T S

The author wishes to thank the following companies and individuals for their assistance in creating this book: ATI Corporation, Borland International Corporation, Microsoft Corporation (Alex St. John and Robert Hess), The A-team at Sapphire Productions (Dan, David, and Matt), Dwayne Evans, Brian Goble, Keith Rupp, Brad Timmins, Emily Timmins.



An Introduction to DirectDraw

Microsoft's DirectDraw is the most exciting tool for PC game programmers since the introduction of the SVGA card. With DirectDraw running under Windows 95, blazingly fast games can be created that rival or even exceed the best game console titles. Simply put: it's here, it's good, and it's fast!

TOPICS COVERED IN THIS BOOK

DirectDraw Programming is a comprehensive C/C++ programmer's tutorial covering the DirectDraw graphics library. Aimed at the intermediate to advanced C/C++ programmer, *DirectDraw Programming* is organized into thirteen chapters:

Chapter One: An Introduction to DirectDraw, provides a brief overview of PC graphics programming and how DirectDraw widens the PC game programmer's horizons.

Chapter Two: How to Get the Most from this Book, details the system requirements of DirectDraw and the organization of the sample programs provided in the book.

Chapter Three: DirectDraw Concepts, covers the interaction of DirectDraw system components: The DirectDraw Software Interface, the DirectDraw HAL, and the COM object interface.

Chapter Four: The DirectDraw Object, introduces the reader to basic DirectDraw programming concepts by creating a small DirectDraw program that prints information about your video card to a window.

Chapter Five: Creating Simple DirectDraw Surfaces: The Primary Surface, introduces DirectDraw's drawing surface metaphor and documents the creation of a primary drawing surface.

Chapter Six: Creating DirectDraw Surfaces: The Back Buffer and Page Flipping, expands on the material presented in Chapter 5 to create a full screen, page flipping display.

Chapter Seven: Creating DirectDraw Surfaces: The Off Screen Surface, covers basic foreground animation using off screen drawing surfaces and introduces DirectDraw's blitting capabilities.

Chapter Eight: Blitting: Transparent Blitting, documents the creation of a DirectDraw color key to produce animation with transparency.

Chapter Nine: Blitting: Blt Effects, covers the large number of graphical special effects that DirectDraw makes available.

Chapter Ten: Blitting: DirectDraw Clippers, introduces the reader to the DirectDrawClipper object and discusses various clipping issues.

Chapter Eleven: DirectDraw Color, covers DirectDraw's extensive color capabilities, illustrating the creation of DirectDraw palettes and adjusting for high-color video modes.

Chapter Twelve: Running Windowed, covers DirectDraw's interaction with the GDI and how a DirectDraw program can run "The Windows Way."

Chapter Thirteen: Debugging DirectDraw Programs, discusses solutions and work-arounds to the special problems DirectDraw imposes on debuggers.

In addition, the included CD-ROM contains a fully functional, arcade-style game written under DirectDraw. The game, "Knob," features hi-speed parallax scrolling, sprite and tile animation, and digital audio mixing through DirectSound. The full source code is provided.

PC GRAPHICS, A BRIEF HISTORY

The need for DirectDraw can best be understood by reviewing the ever changing world of PC game development.

In 1987, IBM introduced the VGA video card. Exceeding its predecessors CGA and EGA in every significant detail, VGA quickly became the de facto standard for PC-compatible computers.

For game developers, VGA offered a rich 256-color graphics mode. The famous VGA mode 13H, and its secretive cousin, mode X, allowed the PC game developer to create detailed graphic scenes with enough color to achieve a degree of artistic and photo-realistic freedom.

Although mode 13H has been a workhorse for PC game programmers, its limitations are readily apparent:

1. *Limited resolution.* Mode 13H delivers 320×200 resolution at 256 colors. Mode X generally operates at 320×240 resolution. While this small resolution is satisfactory for many games, it is a severe drawback for highly detailed imagery and games requiring large amounts of screen real estate.
2. *Limited Color.* Mode 13H is palette dependent. That is, all of the colors that can be displayed on screen are drawn from a table of 256 palette storage registers. If one of these palette registers is changed, every pixel mapped to that register will change. Games requiring a large amount of color, especially those using motion video, suffer greatly under Mode 13H.
3. *Limited Speed.* Due to its hardware design, mode 13H doesn't implement page flipping. Page flipping allows the programmer to maintain two or more screen drawing surfaces. Clean, high-speed animation can be achieved under page flipping by displaying one screen surface while the other is being composed. Without page flipping, the programmer is faced with the choice of updating the screen during a brief time interval known as the *vertical blanking period* (slow and CPU intensive) or face the consequences of updating the display without a timing check (faster, but produces various types of tearing and shearing in animated graphics). Mode X can page flip but doesn't allow simple linear memory access to its drawing surfaces, limiting speed and increasing program complexity. Page flipping and linear memory access are central concepts in DirectDraw and will be covered in detail in later chapters.

SUPER VGA

The Super VGA (SVGA) video card represents an attempt to address the resolution and color limitations of the VGA card. SVGA cards feature more video memory, a plethora of high-resolution and high-color modes, and dedicated hardware to quickly move pixels around the screen. For the game programmer, all of these new features are enticing and immediately useful, but until very recently, they were all but unused.

The problem with SVGA is a lack of a hardware standard. While all SVGA cards are compatible with normal VGA modes, each SVGA card manufacturer took a different course in implementing higher resolution and high-color modes. In an effort to eliminate some of the confusion, the Video Electronics Standards

Association (VESA) created a standard protocol for entering and displaying images in SVGA modes. While the VESA standard has proved beneficial, its usefulness to the game programmer has never been fully realized. Early VESA implementations didn't support page flipping or linear memory access, and any special hardware that the SVGA card included was not used by VESA. This leaves the game programmer with difficult choices: support VESA and live with the speed penalty, write hardware drivers for each SVGA video card on the market (a daunting, impractical task), or support the lowest common denominator and use VGA, even though most of today's PCs are capable of more.

ENTER WINDOWS

Microsoft's Windows operating system has always held out tantalizing possibilities for game developers. Through the use of manufacturer-provided hardware drivers, Windows supports a bewildering number of SVGA video cards, sound cards, and almost everything else made for the PC. Windows provides a consistent protected-mode programming environment and a robust API, but until the advent of DirectDraw, Windows' promise as a premier game platform was unfulfilled.

Windows games must deal with the Graphics Device Interface (GDI), which is a library of Windows functions used to generate graphics. The GDI provides functions for bitmap blitting, geometric drawing primitives, and font generation, among others. The GDI is good at generating device-independent graphics for business applications, but suffers in the highly optimizing, hardware-specific environment of games.

WinG for Windows 3.1 and CreateDIBSection for Win32 are two interesting GDI workarounds. They provide the game programmer with a device-dependent bit map that can be used to compose graphic screens. Once the WinG bit map or the DIB section is created, the programmer can build graphics within the bit map and can copy this zone to the screen. The approach is akin to mode 13H and suffers from the similar problems. The build and copy process is generally CPU-intensive and, without the ability to page flip, it produces tearing and shearing.

DIRECTDRAW FEATURES

DirectDraw is a manager of video cards. DirectDraw allows the programmer to use the video card's advanced features while maintaining a healthy distance from

the low-level details of hardware programming. If the video card has high-resolution, high-color modes, DirectDraw will support them. If the video card contains a hardware blitter chip, DirectDraw can take advantage of it. The idea behind DirectDraw is simple. If the video card has it, let's use it.

DirectDraw makes available the following features:

- *Page flipping.* Current SVGA video cards contain multiple megabytes of video memory. DirectDraw makes this memory available in a simple linear fashion. Multiple page flipping surfaces are easily constructed.
- *Access to blit hardware.* During an initialization phase, DirectDraw determines if the video card has any hardware blitters. If a blitter is present, DirectDraw will use it for video move operations. Some advanced blit chips also include hardware stretching. If DirectDraw detects this feature, it will be used.
- *Multiple resolutions and color depths.* During initialization, DirectDraw determines which video modes the current card can support. These modes are made available through DirectDraw and can be switched at any time.
- *Special surface support.* Some video cards can support special types of drawing surfaces. The most common of these are the overlay and the alpha-blending surface. An overlay surface can be placed transparently over another drawing surface, producing spectacular depth effects. An alpha-blending surface is used to change the color intensity of another surface. Alpha blending can produce a number of interesting effects but is most useful in fading routines.
- *Frame rate timing.* DirectDraw can check the vertical blank signal from the video card to ensure that screen updates take place during the brief time when nothing is being painted on screen. This ensures clean updates during scrolling and animation changes.
- *Clip lists.* DirectDraw can maintain internal clip lists. Clip lists are useful for displaying a small portion of a larger graphic.
- *Non Standard Color models.* Most SVGA cards are built to support the RGB color model. The RGB model is well suited for most types of computer graphics, but is not always the best choice, particularly for motion video. If the video card can support non-RBG surfaces, DirectDraw will manage them and make them accessible to the programmer or to other specialized hardware.

- *Software emulation.* If the current video card doesn't support a critical feature in hardware, DirectDraw attempts to emulate that feature. This software emulation is effectively invisible to the programmer. While DirectDraw doesn't try to emulate every video hardware feature, blitting, clip lists, and linear memory access are covered.

How to Get the Most from this Book

The Microsoft Game Software Development Kit (SDK) includes several elements, collectively referred to as DirectX, designed to achieve the same goal: fast and direct access to multimedia hardware without low-level programming. DirectDraw is the graphics component of the SDK. DirectSound is a digital audio mixer that allows the programmer to set up and play multichannel sound effects. DirectPlay is a communications DLL designed to handle multiple player input over serial and network lines.

INSTALLING THE GAME SDK

Installing the Game SDK from CD-ROM couldn't be easier, simply place the disk into your CD-ROM drive and follow the on-screen prompts. The minimum system requirements are a 486 CPU with 4 megabytes of memory, although a Pentium CPU and 8 or more megabytes of memory are preferred. The Game SDK setup utility can usually determine the correct types of drivers to install for the current system.



WARNING

A manual installation of the Game SDK could corrupt important system files. Be sure to take any necessary precautions.

If necessary, the Game SDK can be installed manually, but the process is more complicated. The Game SDK places the following files in the \WIN95\SYSTEM\ directory:

- DirectDraw files:
 - DDHELP.EXE
 - DDRAW.DLL

DDDRAW16.DLL

DDDRAW16.SYM

<DirectDraw video driver files>

■ DirectSound files:

DSHEL.VXD

DSOUND.DLL

DSOUND.SYM

<DirectSound driver files>

■ DirectPlay files:

DPLAY.DLL

DPLAYNET.DLL

The driver files control specific video and sound cards. If you have an ATI Mach 64 video card and a SoundBlaster 16 sound card, for example, you should copy the following drivers to your \SYSTEM\ directory:

■ DirectDraw ATI Mach 64-specific files:

M64DD16.DLL

M64DD32.DLL

ATIM64.DRV

■ DirectSound SB16 specific files:

SB16.SYM

SB16.VXD

SB16SND.DRV

SBFM.DRV

The Game SDK hardware drivers are meant to replace the drivers that ship with Windows 95 and must be installed through the Windows registry.

The rest of the Game SDK files (examples, source code, utilities, etc.) can be placed anywhere on your hard drive.

FINDING A DIRECTDRAW VIDEO DRIVER

DirectDraw's video drivers are usually categorized by the video hardware type. This can be a problem if you know only the make and model of the video card. Table 2.1 lists several popular video cards and the Microsoft-provided DirectDraw drivers that will work with them:

Table 2.1 Video Cards and Compatible DirectDraw Drivers

Driver Type	Manufacturer and Model
ATI Mach 64	ATI Graphics Expression, ATI WinTurbo
ATI Mach 32	ATI Graphics Wonder
Chips and Technologies	Boca Voyager, C & T Super VGA, ELSA Winner 1280, ELSA Winner 2-1280
Cirrus	Boca SVGA, Boca SuperX, Cardinal SVGA, Cirrus SVGA, Cirrus Laptop, Diamond SpeedStar Pro, Genoa VGA24, Spider 32, Spider 64, STB Horizon
CirrusMM	Actix ProStar, Actix ProStar64, Diamond SpeedStar Pro SE, Diamond SpeedStar 64, Genoa WinVga 64, Number 9 Flashpoint 32, Number 9 Flashpoint 64, Orchid Kelvin 64, Orchid Kelvin EZ, STB Horizon Plus, STB Nitro Orchid
S3	Actix GraphicsEngine 32, Actix GraphicsEngine 64, Actix GraphicsEngine Ultra 64, Actix GraphicsEngine Ultra Plus, Diamond Stealth, Diamond Stealth24, Diamond Stealth Pro, Diamond Stealth 64, Diamond Stealth 64 Video, Diamond Stealth SE, DEC SVGA, ELSA Winner 1000, ELSA Winner 1000 AVI, ELSA Winner 1000 Pro, ELSA Winner 1000 Tri, ELSA Winner2000, ELSA Winner2000 Pro, Genoa Phantom 64, Hercules Graphite 64, Hercules Graphite Terminator 64, Hercules Graphite Terminator Pro, Number 9 GXE,

(Continued)

Driver Type	Manufacturer and Model
S3	Number 9 GXE 64, Number 9 GXE Pro, Number 9 9FX Vision330, Number 9 Motion531, Number 9 Motion771, Orchid Fahrenheit 1280 Plus, Orchid Fahrenheit Pro 64, Orchid Fahrenheit VA, Paradise Bahamas, Paradise Barbados, Spider Tarantula 64, STB Pegasus, STB PowerGraphics Pro, STB PowerGraphics VL24
Tseng	Diamond Stealth 32, Genoa Phantom 32, Hercules Dynamite, Hercules Dynamite Pro, STB Ergo MCX, STB LightSpeed, STB MVP 2X, STB MVP 4X
Western Digital	Diamond SpeedStar 24X, DFI WG-600VL, IBM ThinkPad 755cx, Paradise Accelerator VL Plus, Paradise Super VGA, Western Digital 512K

If your video card doesn't appear on the list, try contacting the manufacturer to determine whether a DirectDraw driver is available.

C/C++ COMPILERS SUPPORTED

The example source code provided in this book is designed to work with the following C/C++ compilers:

Microsoft's Visual C++ Version 2.2 or higher

Borland C++ Version 4.51 or higher

Watcom C/C++ Version 10.5 or higher (requires the WIN32 SDK from Microsoft)

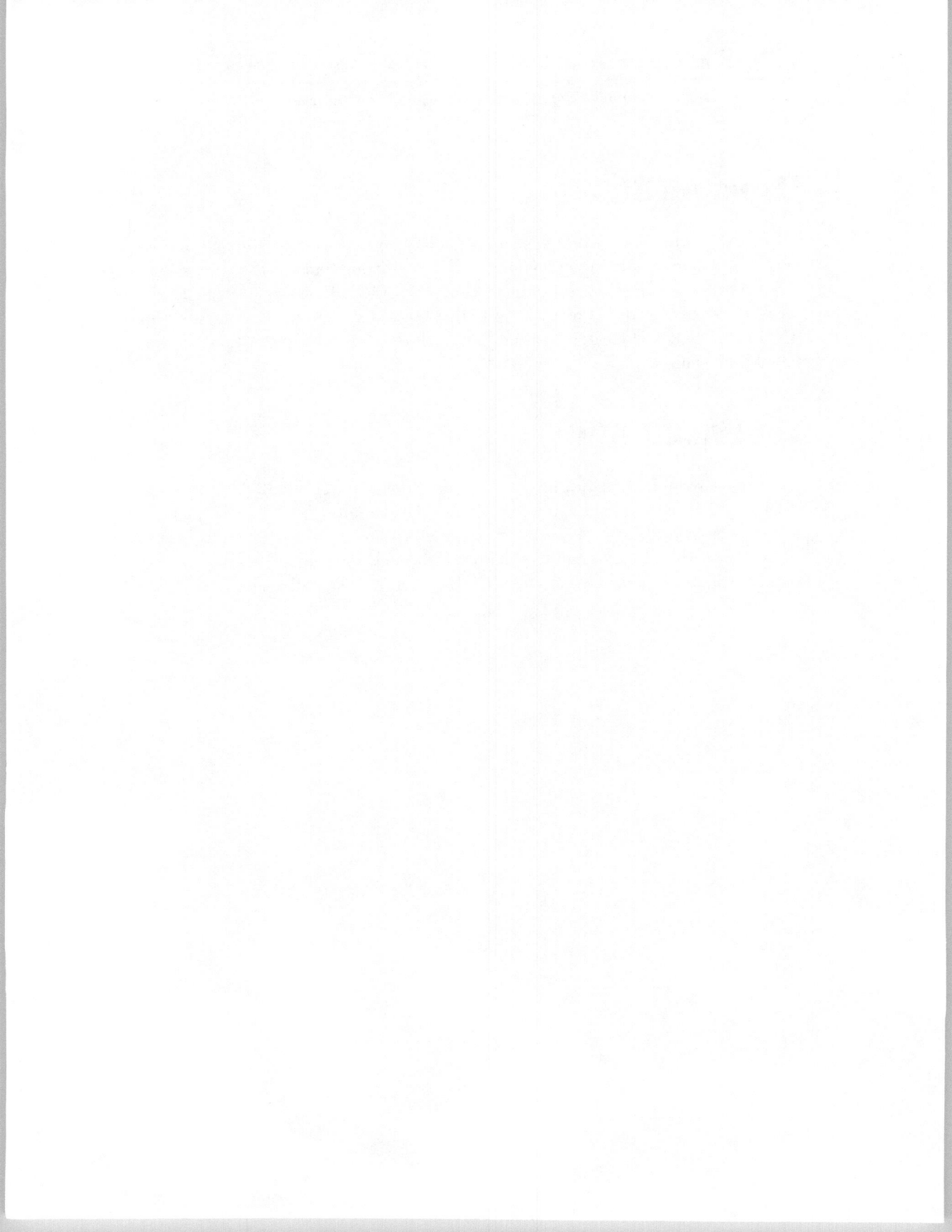
All of the sample programs can be built through the compiler's IDE or at the DOS prompt with a make utility. Borland C++ users must convert the Microsoft supplied DirectX LIBRARY files to a binary format that is compatible with the Borland linker. Detailed information on compiling the example programs can be found in the README.TXT file on the included CD-ROM.

USING ANSI C

All of the source code examples in the book use C++ calling conventions for enacting and manipulating DirectDraw objects and functions. Under this syntax, strict ANSI C-compatible programs cannot be built. Microsoft has included a series of macros in the DirectX INCLUDE files that can be used to simplify the ANSI C syntax for DirectX programs. Consult the GSDK documentation for additional information.

THE WIN32 API

All of the sample DirectDraw programs in the book use the Win32 API for initial window creation, message servicing, and GDI functions. Win32 API functions and their relationships and use in DirectDraw will be explained in the course of the text. Specific API calls can be referenced through your compiler's on-line help system. Additional Windows API texts are listed in the Appendix.



DirectDraw Concepts

Device-independent programming is an essential aspect of Windows. Whether printing a word processor document or writing text to the screen, the programmer is insulated from the actual hardware of the device that is being used. Windows implements this device-independent model with general-purpose function groups (commonly referred to by three-letter acronyms, such as API, GDI, or DDI) and hardware specific device drivers, as shown in Figure 3.1.

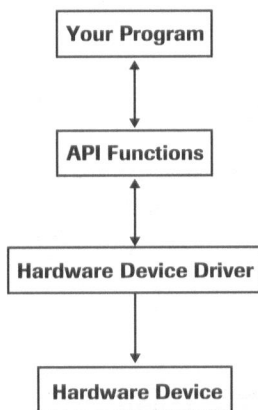


FIGURE 3.1

The API defines a set of frequently used functions for controlling the device. APIs vary greatly in terms of size and complexity. The Windows GDI contains literally hundreds of functions, while a joystick API might contain only two or three.

The types of functions that an API provides can generally be grouped into four broad categories:

- *Initialization.* These functions handle any setup that might be required by the hardware or by the API itself. Typically, the API provides functions for memory allocation, initialization of static variables, and presetting the hardware device to some initial value.
- *Information.* The API provides functions that return information about the current state of API or the hardware device. The information returned can be as simple as a YES/NO flag, or as complex as a series of linked structures.
- *Action.* The core of the API, these functions change the state of the hardware.
- *Cleanup.* These functions free any memory allocated by the API and return the hardware device to a known, “good” state.

The hardware device driver is a set of narrowly focused functions (sometimes referred to as *primitives*) that actually communicate with the hardware device. The device driver, generally provided by the hardware manufacturer, translates the generic functions of the API into specific hardware actions. Once the API has determined that the requested action is legal and appropriate in context, it will call the device driver to implement the action on the current hardware. The API is the only program that can communicate directly with the device driver.

While DirectDraw is built around the API/Hardware Device driver model, it is presented to the programmer as a Component Object Module (COM), or for practical purposes, a C++ object. This means that we must slightly alter our terminology. DirectDraw object functions will be collectively referred to as the DirectDraw software interface and those functions which aren't a part of it will be referred to as DirectDraw API functions.

THE DIRECTDRAW COM

Although DirectDraw COM objects conceptually share much in common with C++ class objects, they are really simple data structures containing pointers to functions. There is nothing about them that is specific to C++. In this way, ANSI C compilers and even non C development systems can take advantage of the an API wrapped up in the COM protocol.

Before using a COM object, it must first be created (instanciated). This returns a pointer to the calling application that contains the address of the COM object structure. The calling application then can make use of the functions contained within the object, often creating additional objects in a fashion similar to C++'s hierarchical class concept. When the calling application is finished using the COM object, it releases it, freeing any memory resources associated with the COM object. For detailed information on the COM interface protocol, consult the Microsoft. White paper, "The Component Object Model Specification."

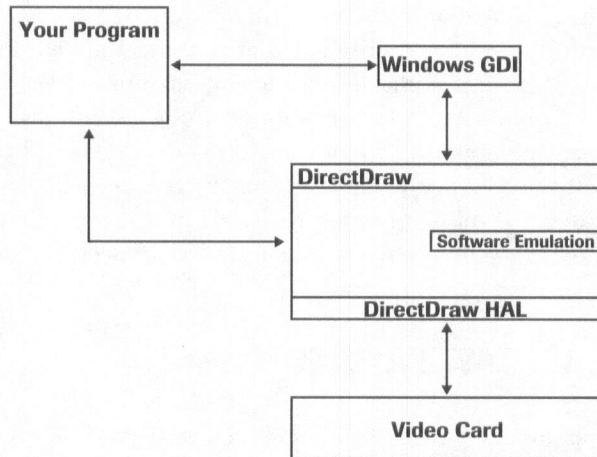
THE DIRECTDRAW MODEL

DirectDraw is composed of three fundamental parts: the DirectDraw software interface, the DirectDraw Hardware Abstraction Layer (DDHAL), and the software emulation layer.

The software interface creates DirectDraw objects, presents information to the programmer, and manages drawing surfaces, among other things. The DirectDraw HAL controls specific hardware devices and is contained in the DirectDraw video driver. The software emulation layer attempts to emulate certain functions that cannot be passed to the hardware through the DirectDraw HAL.

DIRECTDRAW COMMUNICATION

As illustrated in Figure 3.2, all DirectDraw activities flow through the DirectDraw software interface. Once DirectDraw is installed and running, all conventional video operations, including those generated by the GDI, pass through DirectDraw. When a video function is requested, the software interface will determine if the hardware can handle it. If so, the request is passed to the DDHAL. If the function cannot be passed to the DDHAL, the Software Emulation Layer will attempt to replicate the operation using CPU code. An error will be generated when neither the DDHAL nor the software emulation layer can perform a given operation.

**FIGURE 3.2**

THE GDI AND DIRECTDRAW

When DirectDraw is first initialized, it will create a video memory surface for the Windows GDI. This primary surface is shared between the GDI and any DirectDraw programs that may be running. All of the GDI's functions operate normally under DirectDraw. All of the GDI's functions will operate normally under DirectDraw. Programs written for DirectDraw can take advantage of as much or as little of the GDI as they wish.

The GDI will be covered in detail in Chapter 12, "Running Windowed."

The DirectDraw Object

In this chapter, we'll create a small DirectDraw program that connects to the DirectDraw software interface. **CONNECT.CPP** illustrates the creation of a DirectDraw object, setting the mandatory DirectDraw cooperation level, and using a DirectDraw function to determine the hardware capabilities of the current video card.

THE DIRECTDRAWCREATE FUNCTION

The first step in writing a DirectDraw program is to create a DirectDraw object. The DirectDraw object contains data declarations for the current display and member functions for the creation of drawing surfaces, DirectDraw palettes, and DirectDraw clippers. There is one DirectDraw object for each display device (video card) in the system.

A DirectDraw object is created by calling the **DirectDrawCreate** API function:

```
HRESULT DirectDrawCreate(  
    GUID FAR *lpGUID,  
    LPDIRECTDRAW FAR *lplpDD,  
    IUnknown FAR *pUnkOuter )
```

The first parameter, **lpGUID**, is useful only in dual-monitor systems, a topic that is not covered in this book. The normal setting is **NULL**. The second parameter, **lplpDD**, points to a pointer that receives the address of the DirectDraw object. The third parameter, **pUnkOuter**, is for future expansion and should be set to **NULL**.

The return value, **HRESULT**, is used throughout DirectDraw. DirectDraw API and member functions will return the constant **DD_OK** if the operation succeeded. **HRESULT** error codes differ depending on the context of function.

THE SETCOOPERATIVELEVEL FUNCTION

Once the DirectDraw object is successfully created, we can determine how much control our DirectDraw application requires. A game running in full-screen mode shouldn't have to worry about another application destroying its palette or writing to its drawing surfaces. On the other hand, a DirectDraw program running in a window should interact with other programs and allow the GDI to behave in a normal fashion. The **SetCooperativeLevel** member function tells DirectDraw how much control our application should have:

```
HRESULT SetCooperativeLevel ( LPDIRECTDRAW lpDD,
                             HWND hWnd,
                             DWORD dwFlags )
```

lpDD is a pointer to the DirectDraw object we created with the **DirectDrawCreate** function. **hWnd** is a Window handle. The **dwFlags** parameter defines the cooperative level:

DDSCL_ALLOWMODEX

This flag tells direct draw to allow ModeX displays. See **Appendix B: ModeX Surfaces**.

DDSCL_FULLSCREEN

This flag tells DirectDraw that the program will require the entire primary surface area. The **DDSCL_FULLSCREEN** flag only be used with **WS_POPOP** and **WS_TOPMOST** windows that are sized to fill the entire screen.

DDSCL_EXCLUSIVE

If this flag is set, no other application can change the current display mode or alter the current palette. Most of the time, **DDSCL_EXCLUSIVE** is combined with **DDSCL_FULLSCREEN** to allow the creation of page flippable drawing surfaces.

DDSCL_ALLOWREBOOT

This flag allows the user to press the **CRTL_ALT_DEL** key combination when the **DDSCL_FULLSCREEN** and the **DDSCL_EXCLUSIVE** flags are active. If this key combination is pressed, DirectDraw will attempt to free any allocated drawing surfaces, restore the user's video mode, and shut down the DirectDraw program.

DDSCL_NOWINDOWCHANGES

This flag tells DirectDraw not to allow the user to minimize the application's window.

DDSCL_NORMAL

This is used for DirectDraw programs that run in a window and don't require exclusive access to the primary drawing surface or to the palette. Programs using **DDSCL_NORMAL** should act like regular Windows' programs.

THE DIRECTDRAW GETCAPS FUNCTION

The **GetCaps** member function returns vital information about the current display device—its available memory, special hardware, and software emulation capabilities.

```
HRESULT GetCaps ( LPDIRECTDRAW lpDD,  
                  LPDDCAPS lpDDDriverCaps,  
                  LPDDCAPS lpDDHELCaps )
```

lpDD points to the DirectDraw object. **lpDDDriverCaps** is a pointer to a **DDCAPS** structure that the **GetCaps** function will seed with the hardware capabilities of the video card. **lpDDHELCaps** points to a **DDCAPS** structure that receives information about the software emulation capabilities of the DirectDraw driver.

THE DDCAPS STRUCTURE

GetCaps fills the **DDCAPS** structure with important details about the current hardware and what capabilities DirectDraw can emulate through software. A reference to the entire structure can be found in Appendix A: DirectDraw Reference. Several key **DDCAPS** elements are listed there:

DWORD dwCaps

The **dwCaps** element returns any special hardware capabilities that the current video card contains. For software emulation, the return type is identical but indicates that the hardware is not available for this feature; DirectDraw will emulate it through software. **dwCaps** can contain several logically ORed flags. Some significant return types include:

DDCAPS_3D	This flag indicates that the card has special 3D hardware.
DDCAPS_BLT	This flag indicates the video card contains a hardware blitter chip.
DDCAPS_OVERLAY	This flag indicates the video card can support overlaid surfaces.
DDCAPS_NOHARDWARE	This flag indicates that the current video card has no special hardware.
DWORD VidMemTotal	VidMemTotal represents the total amount of video memory, minus any memory that DirectDraw may have allocated during initialization, available to your program.
DWORD VidMemFree	In most DirectDraw programs, many drawing surfaces will be created. The memory requirements of these drawing surfaces cannot always be calculated by multiplying the surface dimensions by the color depth. This value can be used to determine the amount of linear video memory that is available.

THE CONNECT PROGRAM

The following simple program creates a DirectDraw object, sets a DirectDraw cooperation level, and uses the **GetCaps** function to display some information about your video card in a window:

```
//*****
//DirectDraw Programming
//Bret M. Timmins
//
//(C)1995 M&T Books
//
//connect.cpp:
//  A simple example of creating a DirectDraw object
//
//*****

#define WIN32_LEAN_AND_MEAN
#include <windows.h>
#include <windowsx.h>
#if defined( __ ) && defined( __WIN32__ )
```

```

#define _WIN32
#endif
#include <stdio.h>
#include <stdlib.h>
#include <stdarg.h>

LPDIRECTDRAW    lpDirectDrawObject;    // DirectDraw object pointer
DDCAPS          ddcaps;
HRESULT         ConnectToDirectDraw;    // Holds DirectDraw return values

int PASCAL WinMain( HINSTANCE hInstance, HINSTANCE hPrevInstance,
                    LPSTR lpCmdLine, int nCmdShow);
long FAR PASCAL WindowProc ( HWND hwnd, UINT message,
                             WPARAM wParam, LPARAM lParam );
void DisplayDDInfo ( HDC hdc );

/*****
//WinMain - mandatory windows init function
//
*****/
int PASCAL WinMain( HINSTANCE hInstance, HINSTANCE hPrevInstance,
                    LPSTR lpCmdLine, int nCmdShow)
{
    MSG          msg;
    HWND         hwnd;
    WNDCLASS     wc;
    static char  ClassName[] = "Connect";

    lpCmdLine = lpCmdLine;
    hPrevInstance = hPrevInstance;

    //register and realize our display window
    wc.style = CS_HREDRAW | CS_VREDRAW;
    wc.lpfnWndProc = WindowProc;
    wc.cbClsExtra = 0;
    wc.cbWndExtra = 0;
    wc.hInstance = hInstance;
    wc.hIcon = LoadIcon( hInstance, IDI_APPLICATION );
    wc.hCursor = LoadCursor( NULL, IDC_ARROW );
    wc.hbrBackground = GetStockObject( WHITE_BRUSH);
    wc.lpszMenuName = ClassName;
    wc.lpszClassName = ClassName;
    RegisterClass( &wc );

    hwnd = CreateWindow(
        ClassName,
        ClassName,
        WS_OVERLAPPEDWINDOW,
        0,
        0,
        400,

```

```
        400,
        NULL,
        NULL,
        hInstance,
        NULL );

if( !hwnd )
    return FALSE;

ShowWindow( hwnd, nCmdShow );
UpdateWindow( hwnd );
SetFocus( hwnd );

while ( 1 )
{
    if( PeekMessage( &msg, NULL, 0, 0, PM_NOREMOVE ) )
    {
        if( !GetMessage( &msg, NULL, 0, 0 ) )
            return msg.wParam;
        TranslateMessage ( &msg );
        DispatchMessage ( &msg );
    }
    else if( !ActiveApp )
    {
        WaitMessage ();
    }
}

} //WinMain

//*****
//WindowProc - recieve and handle windows messages
//
//
//*****
long FAR PASCAL WindowProc( HWND hwnd, UINT message,
                           WPARAM wParam, LPARAM lParam )
{
    HDC          hdc;
    PAINTSTRUCT ps;

    switch( message )
    {
        case WM_ACTIVATEAPP:
            ActiveApp = wParam;
            break;

        case WM_CREATE:
```

```

//Try to create a DirectDraw object
ConnectToDirectDraw = DirectDrawCreate( NULL,
                                         &lpDirectDrawObject, NULL );

if ( ConnectToDirectDraw == DD_OK )
    lpDirectDrawObject->SetCooperativeLevel( hwnd,
                                             DDSCL_NORMAL );

break;

case WM_KEYDOWN:
    switch( wParam )
    {
        case VK_ESCAPE:
            DestroyWindow( hwnd );
            break;
    }
    break;

case WM_PAINT:
    hdc = BeginPaint ( hwnd, &ps );
    DisplayDDInfo ( hdc );
    EndPaint ( hwnd, &ps );
    return 0L;

case WM_DESTROY:
    if ( ConnectToDirectDraw == DD_OK )
        lpDirectDrawObject->Release ();

    PostQuitMessage( 0 );
    break;

default:
    return DefWindowProc(hwnd, message, wParam, lParam);
}

return 0L;

//WindowProc

//*****
//DisplayDDInfo - print information about current Direct
//                Draw status. Display some flags from
//                the DDCAPS structure
//
//*****
void DisplayDDInfo ( HDC hdc )
{
    TEXTMETRIC  tm;
    char        message[255];

```

```
DDCAPS      ddcaps;
int          TextX = 0;
int          TextY = 0;

GetTextMetrics ( hdc, &tm );

if ( ConnectToDirectDraw == DD_OK )
{
    strcpy ( message, "Connected to DirectDraw" );
    TextOut ( hdc, TextX, TextY, message, strlen (message) );
    TextY += tm.tmHeight+tm.tmExternalLeading;
    strcpy ( message, "DirectDraw Object Created" );
    TextOut ( hdc, TextX, TextY, message, strlen (message) );
    TextY += tm.tmHeight+tm.tmExternalLeading;
    TextY += tm.tmHeight+tm.tmExternalLeading;
}
else
{
    strcpy ( message, "Could not connect to DirectDraw." );
    TextOut ( hdc, 0, 0, message, strlen (message) );
    return;
}

//get DirectDraw hardware capabilities
ddcaps.dwSize = sizeof ( DDcaps );
lpDirectDrawObject->GetCaps ( &ddcaps, NULL );

strcpy ( message, "Total Video Memory: ");
sprintf ( message+strlen("Total Video Memory: "),
          "%i",ddcaps.dwVidMemTotal );
TextOut ( hdc, TextX, TextY, message, strlen (message) );
TextY += tm.tmHeight+tm.tmExternalLeading;

strcpy ( message, "Free Video Memory:");
sprintf ( message+strlen("Free Video Memory: "),
          "%i",ddcaps.dwVidMemFree );
TextOut ( hdc, TextX, TextY, message, strlen (message) );
TextY += tm.tmHeight+tm.tmExternalLeading;
TextY += tm.tmHeight+tm.tmExternalLeading;

strcpy ( message,
        "Your video card contains the following hardware:");
TextOut ( hdc, TextX, TextY, message, strlen (message) );
TextY += tm.tmHeight+tm.tmExternalLeading;

TextX = 16;
if (ddcaps.dwCaps & DDcaps.NOHardware)
{
    strcpy ( message, "No special hardware" );
    TextOut ( hdc, TextX, TextY, message, strlen (message) );
}
```



```

        TextY += tm.tmHeight+tm.tmExternalLeading;
        return;
    }

    if (ddcaps.dwCaps & DDCAPS_BLT)
    {
        strcpy ( message, "A hardware blitter" );
        TextOut ( hdc, TextX, TextY, message, strlen (message) );
        TextY += tm.tmHeight+tm.tmExternalLeading;
    }

    if (ddcaps.dwCaps & DDCAPS_OVERLAY)
    {
        strcpy ( message, "Hardware overlays ");
        TextOut ( hdc, TextX, TextY, message, strlen (message) );
        TextY += tm.tmHeight+tm.tmExternalLeading;
    }

    if (ddcaps.dwCaps & DDCAPS_ALPHA)
    {
        strcpy ( message, "Alpha channels ");
        TextOut ( hdc, TextX, TextY, message, strlen (message) );
        TextY += tm.tmHeight+tm.tmExternalLeading;
    }

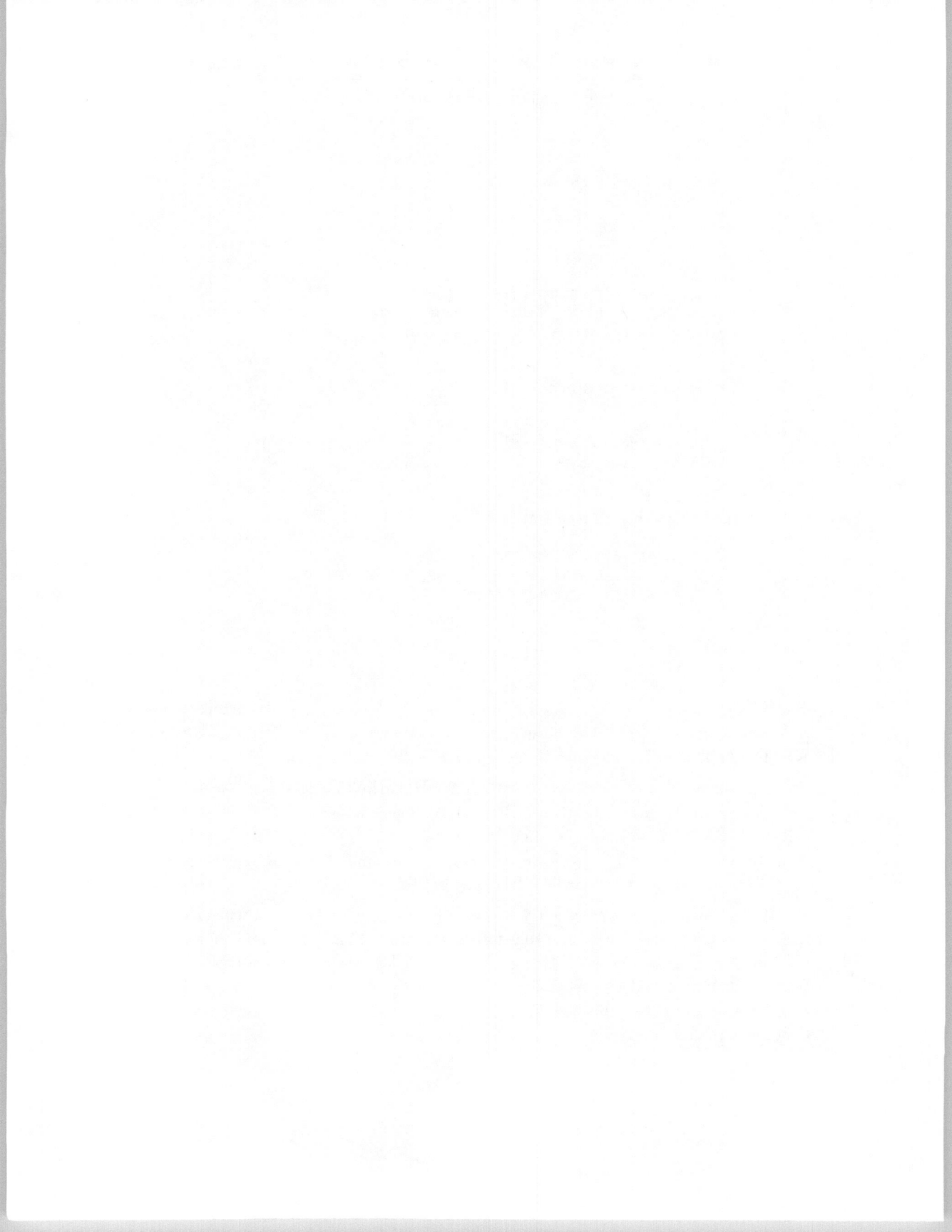
} //DisplayDDInfo

```

The **WinMain** function registers and creates a standard window. In future examples, where the GDI will not be used, our window type will be **WS_POPUP** and **WS_TOPMOST** and our size in **CreateWindow** will be full-screen.

Our **DirectDraw** object is created when a **WM_CREATE** message is received for the window in **WindowProc**. If our **DirectDraw** object was successfully created, we call the **SetCooperativeLevel** member function with our **HWND** handle for this window. Since we want this program to run in a normal Windows fashion, our **DirectDraw** cooperative level will be **DDSCL_NORMAL**.

When our program receives a **WM_PAINT** message, we call the **DisplayDDInfo** function to print some text about the state of our **DirectDraw** object and some information about the video card. If a connection to **DirectDraw** was established in **WindowProc**, we call the **GetCaps** member function with a pointer to a local **DDCAPS** structure. In this function, we only get the hardware information that **GetCaps** returns. The function could easily be extended to check software emulation capabilities.



Creating Simple DirectDraw Surfaces: The Primary Surface

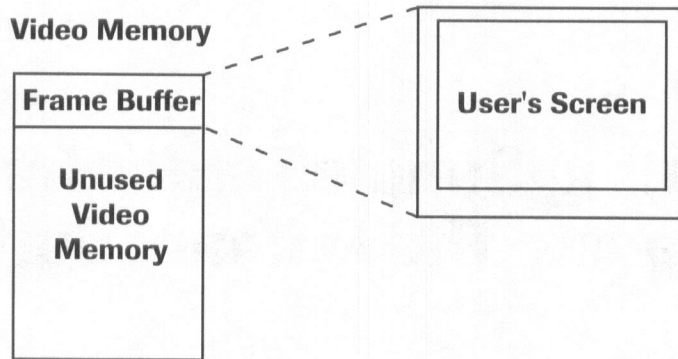
DirectDraw's drawing surfaces are simply areas of memory that hold pixel information. Drawing surfaces can be large—as large as the physical screen and sometimes even bigger—or they can be as small as a single pixel. Surfaces can reside in video memory or in system memory, depending upon the needs of the current application and the capabilities of the hardware. Special drawing surfaces such as overlays and alpha-blending can be created with some types of video cards. In this chapter we will document a fundamental DirectDraw surface type by building a program that creates and fills a primary surface.

THE SURFACE METAPHOR

Bitmapped graphics are generated by setting aside a section of memory to hold pixel data that, when translated by the video hardware, will appear to the user as some sort of meaningful image. On VGA/SVGA systems, this section of memory is contained in the video card itself and is referred to as *video memory*. The section of video memory that contains the current screen image is often called the *frame buffer*.

For a non-DirectDraw Windows program, the frame buffer represents the only area of video memory that can be manipulated by the application. The rest of video memory, as shown in Figure 5.1, remains unused.

DirectDraw makes this hidden video memory accessible as drawing surfaces. In the previous figure, the frame buffer represents the only area of video memory where we can display graphics. In the DirectDraw model, the frame buffer becomes one of many possible drawing surfaces. The number of drawing surfaces that can be created is limited only by the available memory on the video card.

**FIGURE 5.1**

LINEAR MEMORY ACCESS

At a low level, VGA/SVGA memory implementations can be quite intricate. The original VGA standard calls for video memory to be located in a 64 K bank of memory at location A000H. The 64 K bank of memory represents a window in which all video memory must appear. To access the entire piece of video memory that the VGA/SVGA card contains, exotic and rather slow bank-switching schemes must be used.

DirectDraw allows the programmer to access video memory drawing surfaces in a clean, linear fashion, making the entire range of video memory available at any given time to the programmer. In a linear addressing system, simple math can be used to reference any pixel on the drawing surface. A formula for single pixel access becomes:

$$\text{pixel} = Y * \text{surface\_pitch} + X$$

An additional benefit of this system is speed. By eliminating bank-switching code in core graphic-drawing functions, significant speed increases are often realized.

SURFACE PROPERTIES

To create a drawing surface, we must first define the characteristics of the surface. At a minimum, a surface needs to have a width, a height, and a color depth. The width and height specifiers denote the number of pixels that should be set aside for this surface. The color depth (the number of bits required to represent a pixel) of a surface is generally set when Windows initializes DirectDraw but can be altered by changing the current video mode with the **SetDisplayMode** member function.

An important internal element of a surface is its pitch value. The pitch represents the actual amount of memory that is required to store a row of pixels on the surface. The pitch value will always be equal to or greater than the width of the surface. A surface pitch is necessary because some video cards can access their video memory only on evenly aligned boundaries. If a surface has a width of one pixel, for example, its pitch might very well be four if the current video card requires DWORD-aligned surfaces. When initializing a surface with pixel data, it is important to use the surface pitch value instead of the surface width to calculate starting-row positions within the surface.

THE PRIMARY SURFACE

DirectDraw defines a primary surface as any drawing surface that is currently being displayed on screen. A primary surface must be at least as large as the current physical screen dimensions. This means that a surface with a width of 100 pixels and a height of 200 pixels cannot be a primary surface if the user's current screen mode is 640×480 .

In Figure 5.1, we defined a section of video memory that the user is currently seeing, the frame buffer, as the only area of video memory that can be altered by an application. In DirectDraw, the frame buffer and the primary surface are really synonymous, but as stated previously, we can now have as many frame buffers or primary surfaces as our hardware will allow. Figure 5.2 shows several possible primary surface configurations.

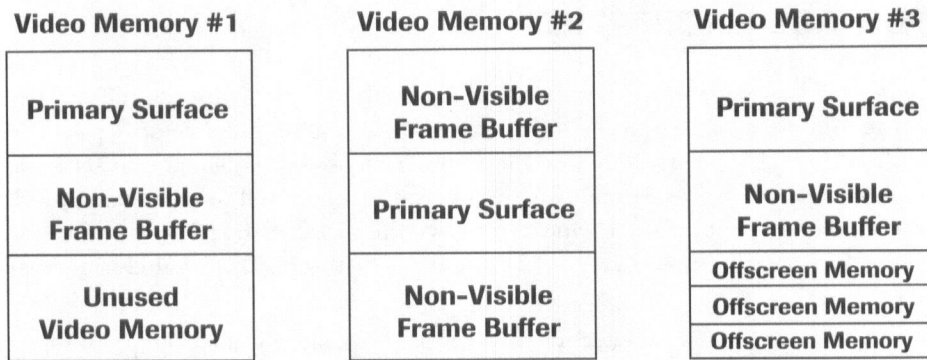


FIGURE 5.2

SURFACE CREATION

To create a drawing surface, we must first establish our DirectDraw object and define our cooperation level with the **SetCooperativeLevel** member function.

We can now call the **CreateSurface** member function to create a drawing surface:

```
HRESULT CreateSurface ( LPDIRECTDRAW lpDD,
                        LPDDSURFACEDESC lpDDSurfaceDesc,
                        LPDIRECTDRAWSURFACE FAR *lpLpDDSurface,
                        IUnknown FAR *pUnkOuter )
```

lpDD points to the DirectDraw object. The second parameter, **lpDDSurfaceDesc**, points to a **DDSURFACEDESC** structure. The values within this structure define the type of surface to be created and its physical dimensions. If the surface was successfully created, the **lpLpDDSurface** pointer will receive the address of a pointer that points to the **DirectDrawSurface** object that represents this surface. The **pUnkOuter** parameter should be set to **NULL**.

It is very important to check the return value of **CreateSurface**. As with all DirectDraw functions, the value **DD_OK** will be returned if the function succeeded. The error flags that **CreateSurface** returns are found in Appendix A, "DirectDraw Reference."

THE DDSURFACEDESC STRUCTURE

DDSURFACEDESC is an input structure that **CreateSurface** uses to determine the type and size of the surface that the programmer wishes to create. Throughout the course of the book, we will expand our knowledge of this structure as new surface types are introduced. A full reference to the **DDSURFACEDESC** structure is found in Appendix A: DirectDraw Reference.

For simple surfaces, the relevant fields of **DDSURFACEDESC** are:

```
DWORD dwFlags  
DWORD dwHeight  
DWORD dwWidth  
DDSCAPS ddscaps
```

DirectDraw uses **dwFlags** to determine which fields within the **DDSURFACEDESC** structure are valid and, by lack of inclusion, which ones should be ignored. DirectDraw defines a series of equates that can be used with the **dwFlags** field. If, for example, you wish to create a surface that requires DirectDraw to look at the **dwHeight**, **dwWidth**, and the **ddscaps** fields, the **dwFlags** field would equal:

```
DDSD_HEIGHT | DDSD_WIDTH | DDSD_CAPS
```

For every field within the **DDSURFACEDESC** structure, there is a corresponding **dwFlags** equate.

The **dwHeight** and **dwWidth** fields define the size of the drawing surface, but as we will see in the **PRIMARY.CPP** program that follows, we don't always need to specify these values. For some surface types, DirectDraw will define them for us.

The **ddscaps** field is a single entry structure that tells DirectDraw what type of surface to create. This is done by setting the **ddscaps** field within the **DDSCAPS** structure. A full reference to surface types is found in Appendix A: DirectDraw Reference." For the following program, our surface type will be **DDSCAPS_PRIMARYSURFACE**.

THE **SetDisplayMode** FUNCTION

When we create a primary surface, we are essentially asking DirectDraw to look at our current screen resolution and color depth, and create a surface based on these characteristics. If our current screen mode is 800×600 at 16 bits of color depth, our primary surface will be sized at 800×600 with 16 bits of color depth. If the current screen mode doesn't suit our needs, we can change it with the **SetDisplayMode** member function:

```
HRESULT SetDisplayMode ( LPDIRECTDRAW lpDD,  
                        DWORD dwWidth,  
                        DWORD dwHeight,  
                        DWORD dwBpp )
```

lpDD points to the DirectDraw object. **dwWidth** and **dwHeight** define the mode size and **dwBpp** is the color depth. **SetDisplayMode** returns **DD_OK** if the video mode was successfully changed. An error return usually indicates that the desired mode is not supported by the video card.

DirectDraw includes a function called **EnumerateDisplayModes** that can be used to determine if a given video mode is supported by the hardware. The **EnumerateDisplayModes** function will be covered in Appendix A. The **PRIMARY.CPP** program that follows assumes that your video card can support 640×480 at 8 bits of color.

THE **RestoreDisplayMode** FUNCTION

When the current screen mode is changed through **SetDisplayMode**, DirectDraw saves the previous mode and allows it to be re-enabled with a simple call to the **RestoreDisplayMode** function:

```
HRESULT RestoreDisplayMode ( LPDIRECTDRAW lpDD )
```

lpDD points to the DirectDraw object. A return of **DD_OK** indicates that the previous video mode was successfully re-entered.

CREATING A PRIMARY SURFACE STEP BY STEP

At this point, we have enough information about surface creation to describe a simple usage scenario for the **PRIMARY.CPP** program:

1. Create a DirectDraw object.
2. Call **SetDisplayMode** to establish the appropriate size and color depth for the primary surface.
3. Set up the **DDSURFACEDESC** structure. Since we are creating a single primary surface, we need only the **ddscaps** field in the **DDSURFACEDESC** structure to be valid:

```
DDSURFACEDESC ddsd;  
  
ddsd.dwSize = sizeof ( ddsd );  
  
ddsd.dwFlags = DDSD_CAPS;  
  
ddsd.ddsCaps.dwCaps = DDSCAPS_PRIMARYSURFACE;
```

4. Call the **CreateSurface** function. If successful, the surface will be created and the **lplpDDSSurface** parameter will point to a **DirectDrawSurface** object that represents this new surface.
5. Initialize the surface with pixel data.
6. At exit, release the primary surface.

THE DIRECTDRAWSURFACE OBJECT

Each surface created in DirectDraw has an associated **DirectDrawSurface** object. The **DirectDrawSurface** object describes the drawing surface and provides member functions for all types of surface activities. The **DirectDrawSurface** object is the mechanism through which surface blitting, page flipping, and surface pointer access, among other things, takes place. A full reference to the **DirectDrawSurface** object is provided in Appendix A: DirectDraw Reference.

In the **PRIMARY.CPP** program, we make use of three **DirectDrawSurface** object member functions: **Lock**, **Unlock**, and **Release**.

THE LOCK FUNCTION

Step five of creating a primary surface requires direct access to the drawing surface's memory to initialize the surface with pixel data. The **Lock** function provides a valid pointer to the surface's memory:

```
HRESULT Lock ( LPDIRECTDRAW lpDDSurface,  
               LPRECT lpDestRect,  
               LPDDSURFACEDESC lpDDSurfaceDesc,  
               DWORD dwFlags,  
               HANDLE hEvent )
```

lpDDSurface is the pointer to the **DirectDrawSurface** object. **lpDestRect** points to a Windows RECT structure. If **lpDestRect** is set to NULL, **Lock** will return a pointer to the top of the drawing surface (0,0). Under this condition, it is assumed that the programmer is requesting exclusive access to the entire drawing surface. If **lpDestRect** points to a valid RECT structure, **Lock** will return a pointer within the surface that equals the address of **RECT.top * surface_pitch + RECT.left**. **lpDDSurfaceDesc** points to a **DDSURFACEDESC** structure that **Lock** will fill with information about the surface. In the **PRIMARY.CPP** program, we will look at two **DDSURFACEDESC** fields:

```
LPVOID lpSurface  
LONG IPitch
```

lpSurface contains the address of the surface memory. The pitch of the surface (the actual amount bytes required to hold a row of pixels) is returned in **IPitch**. **HRESULT** returns **DD_OK** if the surface was successfully locked.

THE UNLOCK FUNCTION

Once we have finished writing to our surface, it is very important to unlock the surface with the **Unlock** member function. When a surface is locked, the application that locked the surface has *exclusive* access to the surface's memory. A

locked surface is inaccessible to DirectDraw itself, and both of DirectDraw's blit functions (**Blt** and **BltFast**) are unusable.

```
HRESULT UnLock ( LPDIRECTDRAW lpDDSurface,  
                LPVOID lpSurfaceData )
```

lpDDSurface is the pointer to the **DirectDrawSurface** object. **lpSurfaceData** is the address of the surface memory that was returned by the **Lock** function in the **DDSURFACEDESC****lpSurface** field. If the entire address space of the surface was locked, **lpSurfaceData** can be set to **NULL**.

THE DIRECTDRAWSURFACE RELEASE FUNCTION

The last DirectDraw function we will need for the **PRIMARY.CPP** program is **Release**. **Release** simply frees drawing surfaces, the **DirectDrawSurface** object and the surface itself, that are no longer needed.

```
DWORD Release ( LPDIRECTDRAW lpDDSurface )
```

lpDDSurface is the pointer to the **DirectDrawSurface** object that is to be freed. The **DWORD** return value can safely be ignored.

THE PRIMARY.CPP PROGRAM

```
//*****  
//DirectDraw Programming  
//Bret Timmins  
//  
//(C)1995 M&T Books  
//  
//primary.cpp  
// Creates and displays a DirectDraw primary surface  
//  
//*****  
#define WIN32_LEAN_AND_MEAN  
#include <windows.h>  
#include <windowsx.h>  
#if defined( __BORLANDC__ ) && defined( __WIN32__ )  
#define _WIN32  
#endif
```

```
#include <ddraw.h>
#include <stdio.h>
#include <stdlib.h>
#include <stdarg.h>

LPDIRECTDRAW          lpDirectDrawObject; //DD object
LPDIRECTDRAWSURFACE   lpPrimary;          //DD primary surface
BOOL                  ActiveApp;           //Is this program active?

int PASCAL WinMain( HINSTANCE hInstance, HINSTANCE hPrevInstance,
                    LPSTR lpCmdLine, int nCmdShow);
long FAR PASCAL WindowProc ( HWND hwnd, UINT message,
                             WPARAM wParam, LPARAM lParam );
BOOL BMPToDirectDrawSurface ( LPDIRECTDRAWSURFACE surface,
                              char *bmp );

//*****
//WinMain - mandatory windows init function
//
//*****
int PASCAL WinMain( HINSTANCE hInstance, HINSTANCE hPrevInstance,
                    LPSTR lpCmdLine, int nCmdShow)
{
    MSG          msg;
    HWND         hwnd;
    WNDCLASS     wc;
    static char  ClassName[] = "Primary";
    DDSURFACEDESC ddsd;
    HRESULT      ddreturn;

    lpCmdLine = lpCmdLine;
    hPrevInstance = hPrevInstance;

    //register and realize our display window
    wc.style = CS_HREDRAW | CS_VREDRAW;
    wc.lpfnWndProc = WindowProc;
    wc.cbClsExtra = 0;
    wc.cbWndExtra = 0;
    wc.hInstance = hInstance;
    wc.hIcon = LoadIcon( hInstance, IDI_APPLICATION );
    wc.hCursor = LoadCursor( NULL, IDC_ARROW );
    wc.hbrBackground = NULL;
    wc.lpszMenuName = ClassName;
    wc.lpszClassName = ClassName;
    RegisterClass( &wc );

    //create a full screen window so that GDI won't ever be
    //called
    hwnd = CreateWindowEx(
```

```
WS_EX_TOPMOST,
ClassName,
ClassName,
WS_POPUP,
0,
0,
GetSystemMetrics(SM_CXSCREEN),
GetSystemMetrics(SM_CYSCREEN),
NULL,
NULL,
hInstance,
NULL );

if( !hwnd )
    return FALSE;

ShowWindow( hwnd, nCmdShow );
UpdateWindow( hwnd );
SetFocus( hwnd );
ShowCursor( FALSE );

//Instantiate our DirectDraw object
ddreturn = DirectDrawCreate( NULL,
                             &lpDirectDrawObject, NULL );
if ( ddreturn != DD_OK )
{
    DestroyWindow( hwnd );
    return FALSE;
}

ddreturn = lpDirectDrawObject->SetCooperativeLevel( hwnd,
                                                    DDSCL_EXCLUSIVE | DDSCL_FULLSCREEN );
if ( ddreturn != DD_OK )
{
    DestroyWindow( hwnd );
    return FALSE;
}

// Set the video mode to 640x480x8
ddreturn = lpDirectDrawObject->SetDisplayMode( 640, 480, 8 );
if ( ddreturn != DD_OK )
{
    DestroyWindow( hwnd );
    return FALSE;
}

// Create the primary surface
ddsd.dwSize = sizeof ( ddsd );
ddsd.dwFlags = DDS_DCAPS;
```

```
ddsd.ddsCaps.dwCaps = DDSCAPS_PRIMARYSURFACE;
ddreturn = lpDirectDrawObject->CreateSurface( &ddsd,
                                              &lpPrimary, NULL );

if ( ddreturn != DD_OK )
{
    DestroyWindow( hwnd );
    return FALSE;
}

//copy a BMP file to the primary surface
if ( !BMPToDirectDrawSurface (lpPrimary, "prim.bmp" ) )
{
    DestroyWindow ( hwnd );
    return FALSE;
}

while ( 1 )
{
    if( PeekMessage( &msg, NULL, 0, 0, PM_NOREMOVE ) )
    {
        if( !GetMessage( &msg, NULL, 0, 0 ) )
            return msg.wParam;
        TranslateMessage ( &msg );
        DispatchMessage ( &msg );
    }
    else if( !ActiveApp )
    {
        WaitMessage ();
    }
}

} //WinMain

//*****
//WindowProc - receive and handle windows messages
//
//
//*****
long FAR PASCAL WindowProc( HWND hwnd, UINT message,
                           WPARAM wParam, LPARAM lParam )
{
    switch( message )
    {
        case WM_ACTIVATEAPP:
            ActiveApp = wParam;
            break;

        case WM_CREATE:
            break;
    }
}
```

```

        case WM_KEYDOWN:
            switch( wParam )
            {
                case VK_ESCAPE:
                    DestroyWindow( hwnd );
                    break;
            }
            break;

        case WM_DESTROY:
            if ( lpDirectDrawObject != NULL )
            {
                if ( lpPrimary != NULL )
                    lpPrimary->Release();
                lpDirectDrawObject->Release();
            }

            ShowCursor( TRUE );
            PostQuitMessage( 0 );
            break;

        default:
            return DefWindowProc (hwnd,message,wParam,lParam);
    }

    return 0L;
}

//WindowProc

//*****
//BMPToDirectDrawSurface - opens and copies a BMP to a DD surface
//input:
// LPDIRECTDRAWSURFACE surface
// char *bmp - filename
//returns:
// BOOL TRUE - success
//*****
BOOL BMPToDirectDrawSurface ( LPDIRECTDRAWSURFACE surface,
                               char *bmp )
{
    HRESULT          ddreturn;
    DWORD            actualRead;
    HANDLE           hfile;
    char             path[MAX_PATH];
    BITMAPFILEHEADER BMPFileHead;
    BITMAPINFOHEADER BMPFileInfo;
    RGBQUAD          Palette[256];
    PALETTEENTRY      pe[256];
    LPDIRECTDRAWPALETTE lpDDPalette;
    DDSURFACEDESC     ddsd;

```

```
int          i,src_width;
BYTE         *lpDDMemory,*lpBMPMemory,*image;

hfile = CreateFile( bmp, GENERIC_READ, FILE_SHARE_READ,
                    (LPSECURITY_ATTRIBUTES)NULL,
                    OPEN_EXISTING, FILE_ATTRIBUTE_NORMAL,
                    (HANDLE) NULL );

if ( hfile == INVALID_HANDLE_VALUE )
{
    strcpy ( path,"..\\");          //back up one directory
    strcat( path, bmp );
    hfile = CreateFile( path, GENERIC_READ, FILE_SHARE_READ,
                        (LPSECURITY_ATTRIBUTES)NULL,
                        OPEN_EXISTING, FILE_ATTRIBUTE_NORMAL,
                        (HANDLE) NULL );
    if ( hfile == INVALID_HANDLE_VALUE )
        return FALSE;
}

// Read the BMP header and info structures
if( !ReadFile ( hfile, &BMPFileHead, sizeof ( BMPFileHead ),
               &actualRead, NULL) )
    return FALSE;

if( !ReadFile ( hfile, &BMPFileInfo, sizeof ( BMPFileInfo ),
               &actualRead, NULL) )
    return FALSE;

//Make sure the BMP is 8-bit color
if ( BMPFileInfo.biBitCount != 8 )
    return FALSE;

//get the BMP palette
if ( !ReadFile ( hfile, Palette, sizeof( Palette ),
               &actualRead, NULL) )
    return FALSE;

//Allocate memory for image data
image = (BYTE *) LocalAlloc ( LPTR,
                              BMPFileInfo.biWidth *
                              BMPFileInfo.biHeight );

if( image == NULL )
    return FALSE;

//read BMP into memory
if( !ReadFile ( hfile, image,
               BMPFileInfo.biWidth * BMPFileInfo.biHeight,
               &actualRead, NULL) )
```



```

{
    LocalFree( image );
    return FALSE;
}

//copy BMP data to DirectDraw surface
memset ( &ddsd, 0, sizeof( DDSURFACEDESC ) );
ddsd.dwSize = sizeof( ddsd );
ddreturn = surface->Lock( NULL, &ddsd, 0, NULL );

if( ddreturn != DD_OK )
{
    LocalFree( image );
    return FALSE;
}

lpDDMemory = (BYTE *) ddsd.lpSurface;
lpBMPMemory = image+((BMPFileInfo.biHeight-1)*
                    BMPFileInfo.biWidth);

if ( BMPFileInfo.biWidth > ddsd.lPitch )
    src_width = ddsd.lPitch;
else
    src_width = BMPFileInfo.biWidth;

for( i = 0; i < BMPFileInfo.biHeight; i++ )
{
    memcpy ( lpDDMemory, lpBMPMemory, src_width );
    lpDDMemory += ddsd.lPitch;
    lpBMPMemory -= src_width;
}

surface->Unlock ( NULL );
LocalFree( image );
CloseHandle(hfile);

//create a DirectDraw palette for the surface
for( i=0; i<256; i++ )
{
    pe[i].peRed = Palette[i].rgbRed;
    pe[i].peGreen = Palette[i].rgbGreen;
    pe[i].peBlue = Palette[i].rgbBlue;
}
ddreturn = lpDirectDrawObject->CreatePalette( DDPCAPS_8BIT,
                                              pe,
                                              &lpDDPalette,
                                              NULL );

if ( ddreturn != DD_OK )
    return FALSE;

```

```
surface->SetPalette ( lpDDPalette );

return TRUE;

} //BMPToDirectDrawSurface
```

In **WinMain**, after registering our window class, we will use the **CreateWindowEx** function for its ability to create a **WS_EX_TOPMOST** window. This extended window style specifies a window that always appears on top of any other GDI window. When we combine **WS_EX_TOPMOST** with **WS_POPUP** and size our window at the maximum screen resolution, we are effectively preventing the GDI from writing to the display. This type of window should be used when building a full-screen DirectDraw application.

If our window was successfully created, we can call the classic **ShowWindow/UpdateWindow** combination to make our window visible. Since we aren't going to handle the **WM_PAINT** message in **WindowProc**, calling **UpdateWindow** is a bit of a formality.

The call to **SetFocus** ensures that our window handler function will receive input from the keyboard.

A mouse pointer isn't needed in the program, so we can hide the mouse with a call to **ShowCursor**.

In the next section of **WinMain**, we create our DirectDraw object and call **SetCooperationLevel** to establish our application as **DDSCL_EXCLUSIVE** and **DDSCL_FULLSCREEN**. **DDSCL_EXCLUSIVE** gives us exclusive use of the color palette and doesn't allow another application to change our screen mode. **DDSCL_FULLSCREEN** tells DirectDraw that we will be using the entire primary surface area.

We then initiate our new display mode and call **CreateSurface** with a local **DDSURFACEDESC** structure initialized to create a primary surface. If there were no errors, we can now write something to our drawing surface.

The **BMPToDirectDrawSurface** function has two input parameters. The first, **LPDIRECTDRAW_SURFACE** surface, is set up to point to our global primary surface object, **lpPrimary**. The second parameter, **bmp**, is simply a character pointer to the path and file name of a windows BMP file.

In a BMP file there are two data structures, **BMPFILEHEADER** and **BMPINFOHEADER**, and a palette (if the BMP graphic has 8 or 4 bits of color) that must be read before we can access the pixel data of the picture. The first part of **BMPToDirectDrawSurface** opens the BMP file and reads in the headers and the palette.

If the file opens successfully and passes the 8-bit color check, we can then use two size fields from the **BMPINFOHEADER** structure to determine the amount of system memory we must set aside to hold the pixel data for this image. We then allocate the required memory with the **LocalAlloc** function and read in pixel information with a call to **ReadFile**.

In the next section of code, we set up a call to the **Lock** function to get a pointer to our DirectDraw surface. If the surface was successfully locked, we can copy the BMP image data to our primary surface. When dealing with BMP files, it is important adjust for the bottom-up orientation of the pixel data. A BMP file is saved with the first row of pixels at the bottom of the file and successive rows of pixels following in descending order. Although this scheme is somewhat odd, we can easily adjust the program to deal with it. The following code fragment gives us the correct starting point within the BMP data zone:

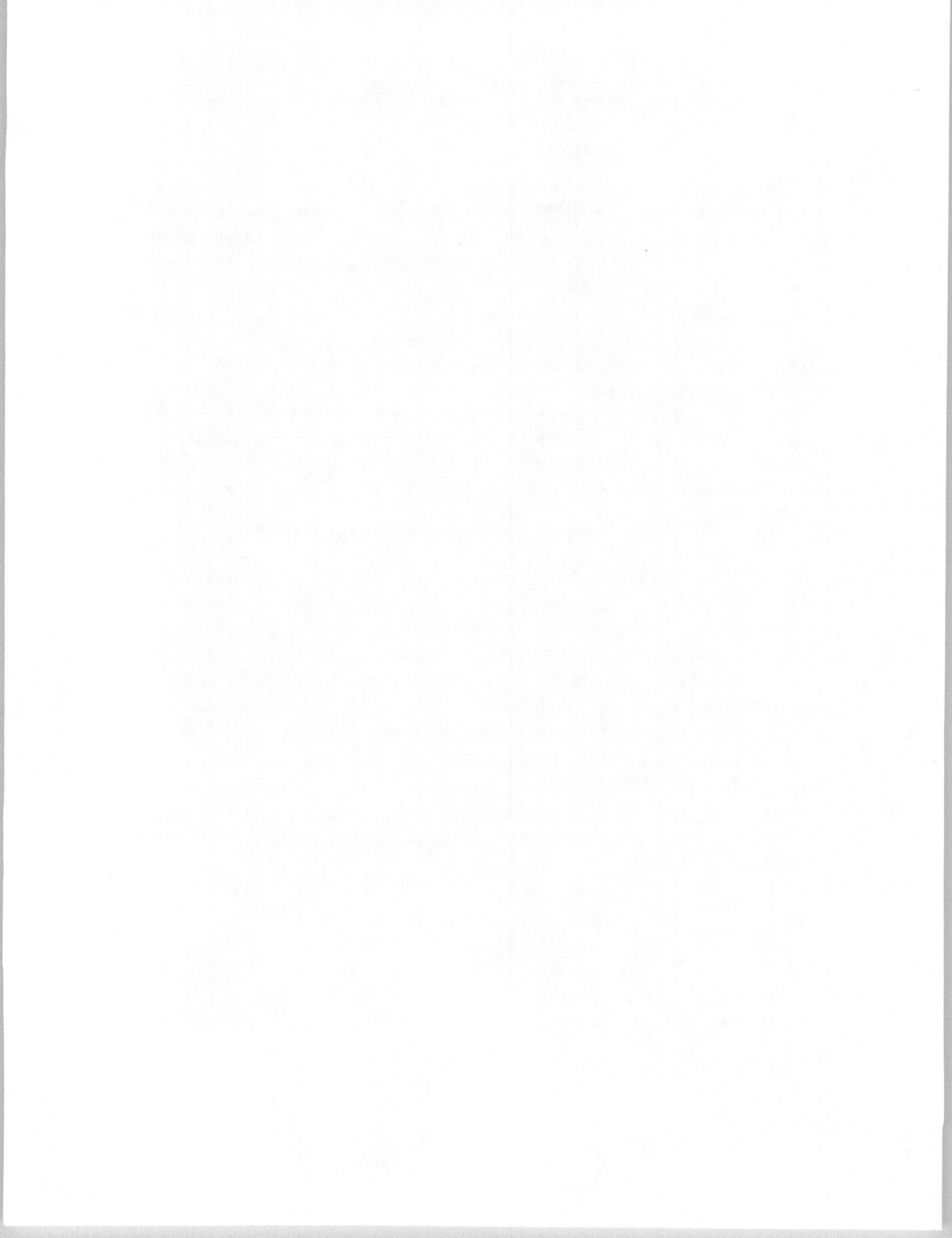
```
lpBMPMemory = image+((BMPFileInfo.biHeight-1)*BMPFileInfo.biWidth);
```

The IF/ELSE clause that follows our BMP pointer adjustment is just a safety check to make sure that we don't copy a BMP that is wider than our primary surface width, a condition that will inevitably lead to a general protection fault error.

The FOR loop copies the BMP data to the primary surface a row of pixels at a time. Within the FOR loop, the DirectDraw surface pointer is incremented by the surface pitch value to take into account surfaces that are aligned on even boundaries and the BMP pointer is decremented to adjust for the bottom-up format of BMP files.

We then **Unlock** our DirectDraw surface and free the memory that was allocated to hold the BMP pixel data.

The last section of **BMPToDirectDrawSurface** creates a DirectDraw palette for our primary surface. DirectDraw palettes and color in general will be covered in detail in Chapter 11.



Creating DirectDraw Surfaces: The Back Buffer and Page Flipping

In Chapter 5, we created a primary drawing surface. A primary surface by itself isn't particularly useful in a game environment since we can't take advantage of the page flipping capabilities at the heart of DirectDraw. In this chapter, we will extend the PRIMARY.CPP program to create two drawing surfaces, a primary surface and a back buffer, and illustrate the page flipping concept.

PAGE FLIPPING

Page flipping (sometimes referred to as *double buffering*) is a simple technique used to achieve clean animation using two or more drawing surfaces as display buffers. The idea is to present a fully composed frame to the user while the program updates the buffer that is currently not visible with the next frame of animation. Once the off-screen frame has been fully drawn, the buffers are switched. The user now sees the new frame and the program begins drawing the next frame of animation on the previously visible surface. Figure 6.1 demonstrates a simple page flip.

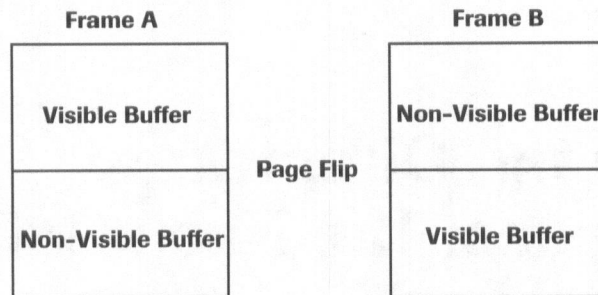


FIGURE 6.1 A SIMPLE PAGE FLIP.

The benefits of page flipping are two-fold. If we synchronize the flipping of our drawing surfaces with the vertical blank-interrupt signal, we can ensure a clean transition from one visible surface to another, totally eliminating any sort of display tearing or shearing that is common in pre-DirectDraw Windows applications. Since we don't have to maintain a copy of our screen in system memory (a technique used in WinG and Mode 13H), our available memory pool becomes larger.

PAGE FLIPPING UNDER DIRECTDRAW

DirectDraw creates an elegant paradigm for page flipping—the attached surface and the flip mechanism. Under DirectDraw, surfaces that are considered attached are dependent upon another drawing surface in some way. In a simple animation environment, where only page flippable buffers are required, surface attachment simply denotes that the drawing surfaces will form a display buffer ring, i.e., one surface will be visible to the user while the other surfaces, the back buffer(s), are being drawn to. The surfaces are then flipped, and drawing continues on the invisible buffers within the ring as displayed in Figure 6.2.

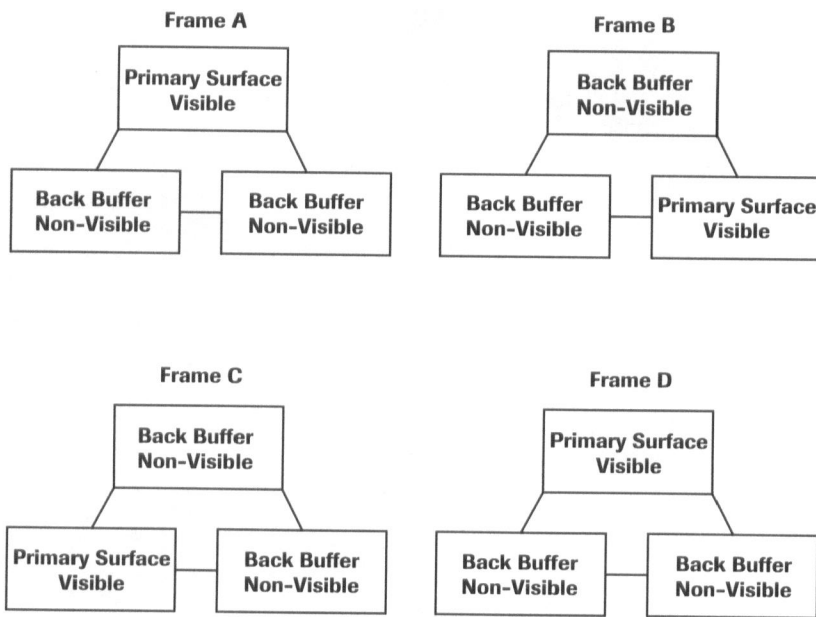


FIGURE 6.2

THE FLIP MECHANISM

In a non-DirectDraw application, the programmer must keep track of which display buffer is currently visible and which buffer(s) can be drawn to for any given frame of animation. DirectDraw automates this process by swapping the pointers to the **DirectDrawSurface** objects when a flip is initiated. From the programmer's standpoint, this means that we can write our graphics to one **DirectDrawSurface** object. The actual surface that will receive our graphics data for a given frame is managed by DirectDraw. This concept is illustrated in Table 6.1. Under this system, we can continuously write our graphics to **SurfaceB**—knowing that it will always be the back buffer in our two buffer animation ring.

TABLE 6.1 THE FLIP MECHANISM

LPDIRECTDRAW SURFACE SurfaceA	LPDIRECTDRAW SURFACE SurfaceB
Frame 1	SurfaceA—The visible surface SurfaceB—The back buffer surface Flip—The pointers to SurfaceA and SurfaceB are swapped
Frame 2	SurfaceA—The visible surface (points to SurfaceB memory) SurfaceB—The back buffer surface (points to SurfaceA memory) Flip—The pointers to SurfaceA and SurfaceB are swapped
Frame 3	Continues the flip cycle

CREATING A BACK BUFFER

In the **BACKBUFF.CPP** program, we need to create two surfaces that fulfill the requirements of a primary surface. As stated in Chapter 5, a primary surface must be as large as the user's physical screen. We can create both of the surfaces with one call to the **CreateSurface** function.

In **PRIMARY.CPP** program, we set up our **DDSURFACEDESC** structure with the following input parameters:

```
DDSURFACEDESC ddsd;

ddsd.dwSize = sizeof ( ddsd );

ddsd.dwFlags = DDSD_CAPS;

ddsd.ddsCaps.dwCaps = DDSCAPS_PRIMARYSURFACE;
```

To create the two attached surfaces we need for this program, we will need to give DirectDraw a little more information about the capabilities of our new surfaces and add a back buffer count to the structure:


```

DDSURFACEDESC ddsd;

ddsd.dwSize = sizeof ( ddsd );

ddsd.dwFlags = DDSD_CAPS | DDSD_BACKBUFFERCOUNT;

ddsd.ddsCaps.dwCaps = DDSCAPS_PRIMARYSURFACE |
                      DDSCAPS_FLIP |
                      DDSCAPS_COMPLEX;

ddsd.dwBackBufferCount = 1;

```

The **dwFlags** field tells DirectDraw that the **ddsCaps.dwCaps** and the **dwBackBufferCount** fields are valid. In the **ddsCaps.dwCaps** field, we define the surface type as primary (**DDSCAPS_PRIMARYSURFACE**). The **DDSCAPS_FLIP** equate allows us to use the **Flip** member function on the surfaces we are creating. **DDSCAPS_FLIP** is a required parameter for surfaces that will be page flipped. The last equate, **DDSCAPS_COMPLEX**, informs DirectDraw that we will be creating multiple surfaces. When **DDSCAPS_COMPLEX** is set, the **dwBackBufferCount** field must be set to one or greater or the call to **CreateSurface** will fail. By setting **dwBackBufferCount** to one, we are creating two surfaces, the primary surface and an attached duplicate surface, which is the back buffer. The number of back buffered surfaces that can be created is limited only by the available video memory, but for practical purposes, one or two back buffers are generally all this is needed.

THE GETATTACHEDSURFACE FUNCTION

When **CreateSurface** is called with this **DDSURFACEDESC** structure, it will return a pointer to the first of the attached surfaces objects. The second surface has been created, but it isn't immediately useable as a drawing surface. We can get a **DirectDrawSurface** object for the attached back buffer (the second surface) by calling the **GetAttachedSurface** member function:

```

HRESULT GetAttachedSurface (
    LPDIRECTDRAWSURFACE lpDDSurface,
    LPDDSCAPS,
    LPLPDIRECTDRAWSURFACE FAR
    *lp1pDDAttachedSurface )

```

lpDDSurface is a pointer to a **DirectDrawSurface** object containing the attached surface. **LPDDSCAPS** points to a **DDSCAPS** structure that tells DirectDraw what type of

attached surface should be returned. For a back buffer, the **DDSCPAS.dwCaps** field should equal **DDSCAPS_BACKBUFFER**. A full list of attached surface equates is found in Appendix A: DirectDraw Reference. The last parameter, **lpDDAttachedSurface**, is the pointer that will receive the address of the **DirectDrawSurface** object for this attached surface. A function return of **DD_OK** indicates success.

THE FLIP FUNCTION

Once we have created our surfaces and obtained a pointer to the attached back buffer surface's object, we can use the Flip member function to do a page flip:

```
HRESULT Flip ( LPDIRECTDRAW lpDDSurface,
               LPDIRECTDRAW lpDDSurfaceTargetOverride
               DWORD dwFlags )
```

lpDDSurface points to a **DirectDrawSurface** object that has been created with the **DDSCAP_FLIP** and the **DDSCAP_FRONTBUFFER** properties. A primary surface (**DDSCAP_PRIMARYSURFACE**) has the **DDSCAP_FRONTBUFFER** flag set by default. The **lpDDSurfaceTargetOverride** parameter can be used to force a specific buffer in a display buffer ring to be the visible surface. By default, DirectDraw flips surfaces in the order they were created (i.e., front buffer, back buffer one, back buffer two, front buffer, etc.). If **lpDDSurfaceTargetOverride** points to a **DirectDrawSurface** object within the display buffer ring, that surface will be the visible surface. **lpDDSurfaceTargetOverride** can be ignored by sending a NULL value. The **dwFlags** parameter can either be set to NULL or **DDFLIP_WAIT**. When Flip is called, the video hardware could be busy with some other drawing activity. The **DDFLIP_WAIT** flag tells DirectDraw to wait until the flip operation can take place before returning. Without the **DDFLIP_WAIT** flag, Flip will return the error code **DDERR_WASSTILLDRAWING** if the flip cannot be immediately performed. Return of **DD_OK** indicates that the surfaces were successfully flipped..

THE BACKBUFF.CPP PROGRAM

```
//*****
//DirectDraw Programming
//Bret M. Timmins
//
//(C)1995 M&T Books
```

```
//
//primary.cpp
//  Creates and displays a DirectDraw primary surface
//  and a backbuffer surface
//*****

#define WIN32_LEAN_AND_MEAN
#include <windows.h>
#include <windowsx.h>
#if defined( __BORLANDC__ ) && defined( __WIN32__ )
#define _WIN32
#endif
#include <ddraw.h>
#include <stdio.h>
#include <stdlib.h>
#include <stdarg.h>

LPDIRECTDRAW          lpDirectDrawObject; // DD object
LPDIRECTDRAWSURFACE lpPrimary;           // DD primary surface
LPDIRECTDRAWSURFACE lpBackbuffer;        // DD back buffer surface
BOOL                  ActiveApp;         //Is this program active?

int PASCAL WinMain( HINSTANCE hInstance, HINSTANCE hPrevInstance,
                    LPSTR lpCmdLine, int nCmdShow);
long FAR PASCAL WindowProc ( HWND hwnd, UINT message,
                             WPARAM wParam, LPARAM lParam );
BOOL BMPToDirectDrawSurface ( LPDIRECTDRAWSURFACE surface,
                              char *bmp );

//*****
//WinMain - mandatory windows init function
//
//*****
int PASCAL WinMain( HINSTANCE hInstance, HINSTANCE hPrevInstance,
                    LPSTR lpCmdLine, int nCmdShow)
{
    MSG          msg;
    HWND         hwnd;
    WNDCLASS     wc;
    static char ClassName[] = "BackBuffer";
    DDSURFACEDESC ddsd;
    DDSCAPS      ddscaps;
    HRESULT      ddreturn;

    lpCmdLine = lpCmdLine;
    hPrevInstance = hPrevInstance;

    //register and realize our display window
    wc.style = CS_HREDRAW | CS_VREDRAW;
    wc.lpfWndProc = WindowProc;
```

```
wc.cbClsExtra = 0;
wc.cbWndExtra = 0;
wc.hInstance = hInstance;
wc.hIcon = LoadIcon( hInstance, IDI_APPLICATION );
wc.hCursor = LoadCursor( NULL, IDC_ARROW );
wc.hbrBackground = NULL;
wc.lpszMenuName = ClassName;
wc.lpszClassName = ClassName;
RegisterClass( &wc );

//create a full screen window so that GDI won't ever be
//called
hwnd = CreateWindowEx(
    WS_EX_TOPMOST,
    ClassName,
    ClassName,
    WS_POPUP,
    0,
    0,
    GetSystemMetrics(SM_CXSCREEN),
    GetSystemMetrics(SM_CYSCREEN),
    NULL,
    NULL,
    hInstance,
    NULL );

if( !hwnd )
    return FALSE;

ShowWindow( hwnd, nCmdShow );
UpdateWindow( hwnd );
SetFocus( hwnd );
ShowCursor( FALSE );

//Instantiate our DirectDraw object

ddreturn = DirectDrawCreate( NULL,
                             &lpDirectDrawObject, NULL );
if ( ddreturn != DD_OK )
{
    DestroyWindow( hwnd );
    return FALSE;
}

ddreturn = lpDirectDrawObject->SetCooperativeLevel( hwnd,
    DDSCL_EXCLUSIVE | DDSCL_FULLSCREEN );
if ( ddreturn != DD_OK )
{
    DestroyWindow( hwnd );
    return FALSE;
}
```

```
}

// Set the video mode to 640x480x8
ddreturn = lpDirectDrawObject->SetDisplayMode( 640, 480, 8);
if ( ddreturn != DD_OK )
{
    DestroyWindow( hwnd );
    return FALSE;
}

// Create the primary surface and one back buffer surface
ddsd.dwSize = sizeof( ddsd );
ddsd.dwFlags = DDSD_CAPS | DDSD_BACKBUFFERCOUNT;
ddsd.ddsCaps.dwCaps = DDSCAPS_PRIMARYSURFACE |
                    DDSCAPS_FLIP |
                    DDSCAPS_COMPLEX;
ddsd.dwBackBufferCount = 1;
ddreturn = lpDirectDrawObject->CreateSurface( &ddsd,
                                             &lpPrimary, NULL );
if ( ddreturn != DD_OK )
{
    DestroyWindow( hwnd );
    return FALSE;
}

//get a surface pointer to our back buffer
ddscaps.dwCaps = DDSCAPS_BACKBUFFER;
ddreturn = lpPrimary->GetAttachedSurface(&ddscaps,
                                         &lpBackbuffer);
if( ddreturn != DD_OK )
{
    DestroyWindow ( hwnd );
    return FALSE;
}

//copy a BMP file to the primary surface
if ( !BMPToDirectDrawSurface (lpPrimary, "prim.bmp" ) )
{
    DestroyWindow ( hwnd );
    return FALSE;
}

//copy to the back buffer
if ( !BMPToDirectDrawSurface (lpBackbuffer, "back.bmp" ) )
{
    DestroyWindow ( hwnd );
    return FALSE;
}

while ( 1 )
```

```
{
    if( PeekMessage( &msg, NULL, 0, 0, PM_NOREMOVE ) )
    {
        if( !GetMessage( &msg, NULL, 0, 0 ) )
            return msg.wParam;
        TranslateMessage ( &msg );
        DispatchMessage ( &msg );
    }
    else if( !ActiveApp )
    {
        WaitMessage ();
    }
}

} //WinMain

//*****
//WindowProc - receive and handle windows messages
//
//
//*****
long FAR PASCAL WindowProc( HWND hwnd, UINT message,
                           WPARAM wParam, LPARAM lParam )
{
    HRESULT ddreturn;

    switch( message )
    {
        case WM_ACTIVATEAPP:
            ActiveApp = wParam;
            break;

        case WM_CREATE:
            break;

        case WM_KEYDOWN:
            switch ( wParam )
            {
                case VK_SPACE:
                    ddreturn=lpPrimary->Flip( NULL,DDFLIP_WAIT);

                    if ( ddreturn == DDERR_SURFACELOST )
                    {
                        ddreturn = lpPrimary->Restore();
                        if ( ddreturn == DD_OK )
                        {
                            if (!BMPToDirectDrawSurface(
                                lpPrimary,
                                "prim.bmp"))
                                {

```

```
        DestroyWindow ( hwnd );
        break;
    }
}
ddreturn = lpBackbuffer->Restore();
if( ddreturn == DD_OK )
{
    if ( !BMPToDirectDrawSurface (
        lpBackbuffer,
        "back.bmp" ))
    {
        DestroyWindow ( hwnd );
        break;
    }
}

if ( ddreturn != DD_OK )
    DestroyWindow ( hwnd );
break;

case VK_ESCAPE:
    DestroyWindow ( hwnd );
    break;
}
break;

case WM_DESTROY:
    if ( lpDirectDrawObject != NULL )
    {
        if ( lpBackbuffer != NULL )
            lpBackbuffer->Release();
        if ( lpPrimary != NULL )
            lpPrimary->Release();
        lpDirectDrawObject->Release();
    }

    ShowCursor( TRUE );
    PostQuitMessage( 0 );
    break;

default:
    return DefWindowProc (hwnd,message,wParam,lParam);
}

return 0L;

} //WindowProc

//*****
```

```
//BMPToDirectDrawSurface - opens and copies a BMP to a DD surface
//input:
// LPDIRECTDRAWSURFACE surface
// char *bmp - filename
//returns:
// BOOL TRUE - success
//*****
BOOL BMPToDirectDrawSurface ( LPDIRECTDRAWSURFACE surface,
                               char *bmp )
{
    HRESULT          ddreturn;
    DWORD            actualRead;
    HANDLE           hfile;
    char             path[MAX_PATH];
    BITMAPFILEHEADER BMPFileHead;
    BITMAPINFOHEADER BMPFileInfo;
    RGBQUAD          Palette[256];
    PALETTEENTRY      pe[256];
    LPDIRECTDRAWPALETTE lpDDPalette;
    DDSURFACEDESC    ddsd;
    int              i,src_width;
    BYTE             *lpDDMemory,*lpBMPMemory,*image;

    hfile = CreateFile( bmp, GENERIC_READ, FILE_SHARE_READ,
                        (LPSECURITY_ATTRIBUTES)NULL,
                        OPEN_EXISTING, FILE_ATTRIBUTE_NORMAL,
                        (HANDLE) NULL );

    if ( hfile == INVALID_HANDLE_VALUE )
    {
        strcpy (path, "..\\");          //back up one directory
        strcat( path, bmp );
        hfile = CreateFile( path, GENERIC_READ, FILE_SHARE_READ,
                            (LPSECURITY_ATTRIBUTES)NULL,
                            OPEN_EXISTING, FILE_ATTRIBUTE_NORMAL,
                            (HANDLE) NULL );
        if ( hfile == INVALID_HANDLE_VALUE )
            return FALSE;
    }

    // Read the BMP header and info structures
    if( !ReadFile ( hfile, &BMPFileHead, sizeof ( BMPFileHead ),
                    &actualRead, NULL) )
        return FALSE;

    if( !ReadFile ( hfile, &BMPFileInfo, sizeof ( BMPFileInfo ),
                    &actualRead, NULL) )
        return FALSE;

    //Make sure the BMP is 8-bit color
```



```
if ( BMPFileInfo.biBitCount != 8 )
    return FALSE;

//get the BMP palette
if ( !ReadFile ( hfile, Palette, sizeof( Palette ),
                &actualRead, NULL ) )
    return FALSE;

//Allocate memory for image data
image = (BYTE *) LocalAlloc ( LPTR,
                             BMPFileInfo.biWidth *
                             BMPFileInfo.biHeight );

if( image == NULL )
    return FALSE;

//read BMP into memory
if( !ReadFile ( hfile, image,
               BMPFileInfo.biWidth * BMPFileInfo.biHeight,
               &actualRead, NULL ) )
{
    LocalFree( image );
    return FALSE;
}

//copy BMP data to DirectDraw surface
memset ( &ddsd, 0, sizeof( DDSURFACEDESC ) );
ddsd.dwSize = sizeof( ddsd );
ddreturn = surface->Lock( NULL, &ddsd, 0, NULL );

if( ddreturn != DD_OK )
{
    LocalFree( image );
    return FALSE;
}

lpDDMemory = (BYTE *) ddsd.lpSurface;
lpBMPMemory = image+((BMPFileInfo.biHeight-1)*
                    BMPFileInfo.biWidth);

if ( BMPFileInfo.biWidth > ddsd.lPitch )
    src_width = ddsd.lPitch;
else
    src_width = BMPFileInfo.biWidth;

for( i = 0; i < BMPFileInfo.biHeight; i++ )
{
    memcpy ( lpDDMemory, lpBMPMemory, src_width );
    lpDDMemory += ddsd.lPitch;
    lpBMPMemory -= src_width;
}
```

```

surface->Unlock ( NULL );
LocalFree( image );
CloseHandle(hfile);

//create a DirectDraw palette for the surface
for( i=0; i<256; i++ )
{
    pe[i].peRed = Palette[i].rgbRed;
    pe[i].peGreen = Palette[i].rgbGreen;
    pe[i].peBlue = Palette[i].rgbBlue;
}
ddreturn = lpDirectDrawObject->CreatePalette( DDPCAPS_8BIT,
                                              pe,
                                              &lpDDPalette,
                                              NULL );

if ( ddreturn != DD_OK )
    return FALSE;
surface->SetPalette ( lpDDPalette );

return TRUE;

} //BMPToDirectDrawSurface

```

In the **WinMain** function, we establish our DirectDraw object, set our cooperation level, and set the display mode. We then set up a **DDSURFACEDESC** structure with the appropriate values to create a primary surface with an attached back buffer.

After the call to **CreateSurface**, we have a pointer to the primary surface object, **lpPrimary**. The call to **GetAttachedSurface** retrieves a **DirectDrawSurface** object for the back buffer, which we put in **lpBackbuffer**. Both surfaces are then initialized with pixel data through the **BMPToDirectDrawSurface** function.

The page flip takes place in the **WindowProc** handler when the **Spacebar** is pressed.

This program simply flips between two attached surfaces. In Chapter 7, we will more fully exploit page flipping by dynamically blitting pixel data to the alternating surfaces.

Creating DirectDraw Surfaces: The Off-Screen Surface

In a typical game environment, we can break down the memory that is used for graphics into two fundamental parts. The first part is memory that is dedicated to the display. In mode 13H, for example, we have the 64K section of video memory that makes up the display itself. A mode 13H game might also allocate another 64K section of system memory as a display buffer to fully and cleanly compose a frame of graphics. As we have seen in the last two chapters, DirectDraw gives us more robust tools for our display memory needs. We can create a primary surface and one or more back buffers, and flip between them to achieve glitch-free transitions from one visible surface to another.

The second aspect of a game's memory allocation involves setting aside sections of memory to hold predefined imagery. These predefined images can be animation sets, backgrounds, fonts, status bars, or anything else that the game requires. This type of graphics memory differs conceptually from display memory in that the user never directly sees these images in their native memory locations, but rather sees a copy of them when they are "blted" to the currently visible display surface. DirectDraw defines this type of memory as an off-screen drawing surface.

In this chapter, we will extend the BACKBUFF.CPP program to incorporate off-screen surfaces and take advantage of DirectDraw's blting capabilities.

BEING OFF SCREEN

Off-screen surfaces have a programmer-determined height, width, and pixel depth. As with primary and back-buffer surfaces, the pixel depth of an off-screen

surface is initially determined by DirectDraw (the user's current video mode) but can be overridden with a call to the **SetDisplayMode** function. The physical dimensions of an off-screen surface can be as small as a single pixel or as large as the video card and available memory will allow.



NOTE

Some video cards cannot create surfaces that are larger than the dimensions of the current video mode, regardless of the amount of video memory that the card might contain. In general, it is a good idea not to create surfaces that are larger than the primary surface.

Once the off-screen surface has been created, it is accessed in exactly the same manner as primary and back buffer surfaces. That is, a pointer to the off-screen surface memory is obtained through the Lock/Unlock mechanism, an additional, and important return of the **Lock** function is the pitch of the off-screen surface.

CREATING AN OFF-SCREEN SURFACE

As with all DirectDraw surfaces, an off-screen surface is created by setting the appropriate fields within a **DDSURFACEDESC** structure and then calling the **CreateSurface** function:

```
DDSURFACEDESC ddsd;
ddsd.dwSize = sizeof ( ddsd );
ddsd.dwFlags = DDSD_CAPS | DDSD_HEIGHT | DDSD_WIDTH;
ddsd.ddsCaps.dwCaps = DDSD_OFFSCREENPLAIN;
ddsd.dwHeight = (Surface Height);
ddsd.dwWidth = (Surface Width);
```

The **ddsd.dwFlags** field informs DirectDraw which fields within the **ddsd** structure should be considered valid for the creation of this surface. The surface capability equate, **DDSD_OFFSCREENPLAIN**, tells DirectDraw that the surface will be off screen and that it has no special characteristics; that is, this surface is just allocated memory. The **ddsd.dwHeight** and **ddsd.dwWidth** fields define the dimensions of the surface in pixels.

By default, DirectDraw will attempt to create off screen surfaces in an unused section of video memory. Most of the time, this is exactly where we want our off screen surfaces to be. We can, however, tell DirectDraw to create off screen drawing surfaces in system memory.

THE SYSTEM MEMORY SURFACE

Off screen surfaces can reside in video memory or they can be created to exist in the computer's system memory. System memory surfaces are accessed in exactly the same manner as video memory surfaces, i.e., they can be "locked" to acquire a valid surface pointer and are fully compatible with the blt functions. System memory surfaces are created by explicitly requesting them with the **DDSD_SYSTEMMEMORY** flag:

```
DDSURFACEDESC ddsd;
ddsd.dwSize = sizeof ( ddsd );
ddsd.dwFlags = DDSD_CAPS | DDSD_HEIGHT | DDSD_WIDTH;
ddsd.ddsCaps.dwCaps = DDSD_OFFSCREENPLAIN
| DDSD_SYSTEMMEMORY;
ddsd.dwHeight = (Surface Height);
ddsd.dwWidth = (Surface Width);
```

While system memory surfaces can't take advantage of the hardware blter (blting to and from a system memory surface will take place through DirectDraw's software emulation code), system memory surfaces can actually offer a performance increase for some types of programs, depending on the current CPU and video card (DRAM vs. VRAM etc.). In a 3D rendering environment, where every pixel must be computed, it might be faster to render to a system memory surface and then blt the fully composed screen to a video memory back buffer. As with all speed-critical programs, testing at runtime will determine the best solution.

System memory surfaces also provide a mechanism for debugging DirectDraw program when using a GDI based debugger. See Chapter 13: Debugging DirectDraw Programs for details.

Video memory surfaces can also be explicitly requested using the **DDSD_VIDEOMEMORY** flag. As stated above, DirectDraw will attempt to create off screen surfaces in video memory when using a lone **DDSD_OFFSCREENPLAIN** flag. If there isn't enough unused video memory available for the drawing surface, DirectDraw will allocate the surface memory from the system memory pool. The **DDSD_VIDEOMEMORY** flag tells DirectDraw to only attempt to create the surface in video memory. If there isn't enough video memory for the surface, DirectDraw will return an error. The DirectDraw program can then adjust its surface memory requirements and try to re-create the drawing surfaces (at a smaller size, for example) or simply inform the user that the current video memory isn't sufficient and exit.

BLTING

The term *blitting* simply refers to moving pixel data from one memory location to another. A blt can be executed by the programmer with CPU code, or it can take place through special hardware contained on the video card, commonly known as a blter (or blitter) chip. A blt operation can be as simple as a linear pixel to pixel move or it can be as complicated as a 3D pixel transformation with transparency. Whatever method is used, the idea is to move pixels from their source location to the desired destination surface as fast as possible.

DirectDraw gives us two blting functions: **Blt** and **BltFast**. In the OFFSCREEN.CPP program, we will be using **BltFast**. The slightly more complicated **Blt** functions will be covered in Chapter 9: Blting: Blt Effects.

THE BLTFAST FUNCTION

As its name implies, the **BltFast** function attempts to blt pixels from one surface to another as quickly as it can. **BltFast** can't use DirectDraw clipper objects, so it's up to the programmer to ensure that the blt is within a legal range. If a hardware blter is present and both surfaces reside in video memory **Bltfast** will execute the blt operation through the hardware for an extremely quick pixel transfer. If the current video card doesn't contain a blter, the blt operation will take place through DirectDraw's software emulation.

```
HRESULT BltFast ( LPDIRECTDRAWSURFACE lpDDSurface,
    DWORD dwX,
    DWORD dwY,
    LPDIRECTDRAWSURFACE lpDDSrcSurface,
    LPRECT lpSrcRect,
    DWORD dwTrans )
```

The **lpDDSurface** parameter points to the surface that will be the destination of the blt. **dwX** and **dwY** define the starting locations of the blt within the destination surface. **lpDDSrcSurface** is a pointer to the source surface object. The **lpSrcRect** parameter points to a standard Windows RECT structure that defines what portion of the source surface should be blted to the destination surface. The last parameter, **dwTrans**, specifies the type of blt transfer that will occur. If **dwTrans** is set to **DDBLT_NOCOLORKEY**, the blt operation will be a straight copy of the pixels from one surface to the other. A value of **DDBLT_SRCCOLORKEY** tells DirectDraw that blt operation should take place with some of the pixels in the source surface being

treated as transparent and not copied to the destination surface. Transparent blting will be covered in Chapter 8t. A return value of **DD_OK** indicates a successful blt.

As an example, if we wish to set up a blt between an off-screen surface that is sized at 100 × 100 and a back-buffer surface, our source code might look like this:

```
LPDIRECTDRAWSURFACE lpOffscreenSurface;
LPDIRECTDRAWSURFACE lpBackBufferSurface;
RECT SrcRect;
SrcRect.Left = 0;
SrcRect.top = 0;
SrcRect.right = 100;
SrcRect.bottom = 100;
lpBackBufferSurface->BltFast ( 0, 0,

    lpOffscreenSurface,
    &SrcRect,
    DDBLT_NOCOLORKEY );
```

In this example, we would be blting all of the pixels in the off-screen surface to the back buffer beginning at the back buffer's first pixel location (0,0). To blt only the bottom half of the off-screen surface to the back buffer, we would change the RECT structure to equal:

```
SrcRect.left = 50;
SrcRect.top = 50;
SrcRect.right = 100;
SrcRect.bottom = 100;
```

THE OFF_SCR.CPP PROGRAM

```
//*****
//The DirectDraw Developer's Guide
//Bret Timmins
//
//(C)1995 M&T Books
//
//off_scr.cpp
// Demonstrates blting an offscreen surface in a
// page flipping enviroment
//*****
#define WIN32_LEAN_AND_MEAN
#include <windows.h>
#include <windowsx.h>
#if defined( __BORLANDC__ ) && defined( __WIN32__ )
#define _WIN32
```



```
#endif
#include <ddraw.h>
#include <stdio.h>
#include <stdlib.h>
#include <stdarg.h>

LPDIRECTDRAW          lpDirectDrawObject; //DirectDraw object
LPDIRECTDRAWSURFACE lpPrimary;           //DD primary surface
LPDIRECTDRAWSURFACE lpBackbuffer;        //DD back buffer
LPDIRECTDRAWSURFACE lpOffscreen;         //offscreen surface
BOOL                  ActiveApp;          //Is this program active?

int PASCAL WinMain( HINSTANCE hInstance, HINSTANCE hPrevInstance,
                    LPSTR lpCmdLine, int nCmdShow);
long FAR PASCAL WindowProc ( HWND hwnd, UINT message,
                             WPARAM wParam, LPARAM lParam );

BOOL DrawFrame ( void );
BOOL RestoreSurfaces ( void );
BOOL BMPToDirectDrawSurface ( LPDIRECTDRAWSURFACE surface,
                              char *bmp );

//*****
//WinMain - mandatory windows init function
//
//*****
int PASCAL WinMain( HINSTANCE hInstance, HINSTANCE hPrevInstance,
                    LPSTR lpCmdLine, int nCmdShow)
{
    MSG          msg;
    HWND         hwnd;
    WNDCLASS     wc;
    static char  ClassName[] = "Offscreen";
    DDSURFACEDESC ddsd;
    DDSCAPS      ddscaps;
    HRESULT      ddreturn;

    lpCmdLine = lpCmdLine;
    hPrevInstance = hPrevInstance;

    //register and realize our display window
    wc.style = CS_HREDRAW | CS_VREDRAW;
    wc.lpfnWndProc = WindowProc;
    wc.cbClsExtra = 0;
    wc.cbWndExtra = 0;
    wc.hInstance = hInstance;
    wc.hIcon = LoadIcon( hInstance, IDI_APPLICATION );
    wc.hCursor = LoadCursor( NULL, IDC_ARROW );
    wc.hbrBackground = NULL;
    wc.lpszMenuName = ClassName;
    wc.lpszClassName = ClassName;
```



```
RegisterClass( &wc );

//create a full screen window so that GDI won't ever be
//called
hwnd = CreateWindowEx(
    WS_EX_TOPMOST,
    ClassName,
    ClassName,
    WS_POPUP,
    0,
    0,
    GetSystemMetrics(SM_CXSCREEN),
    GetSystemMetrics(SM_CYSCREEN),
    NULL,
    NULL,
    hInstance,
    NULL );

if( !hwnd )
    return FALSE;

ShowWindow( hwnd, nCmdShow );
UpdateWindow( hwnd );
SetFocus( hwnd );
ShowCursor( FALSE );

//Instanciate our DirectDraw object
ddreturn = DirectDrawCreate( NULL,
                             &lpDirectDrawObject, NULL );
if ( ddreturn != DD_OK )
{
    DestroyWindow( hwnd );
    return FALSE;
}

ddreturn = lpDirectDrawObject->SetCooperativeLevel( hwnd,
                                                    DDSCS_EXCLUSIVE | DDSCS_FULLSCREEN );
if ( ddreturn != DD_OK )
{
    DestroyWindow( hwnd );
    return FALSE;
}

// Set the video mode to 640x480x8
ddreturn = lpDirectDrawObject->SetDisplayMode( 640, 480, 8 );
if ( ddreturn != DD_OK )
{
    DestroyWindow( hwnd );
    return FALSE;
}
```

```
// Create the primary surface and one back buffer surface
ddsd.dwSize = sizeof( ddsd );
ddsd.dwFlags = DDSD_CAPS | DDSD_BACKBUFFERCOUNT;
ddsd.ddsCaps.dwCaps = DDSCAPS_PRIMARYSURFACE |
                    DDSCAPS_FLIP |
                    DDSCAPS_COMPLEX;
ddsd.dwBackBufferCount = 1;
ddreturn = lpDirectDrawObject->CreateSurface( &ddsd,
                                              &lpPrimary, NULL );

if ( ddreturn != DD_OK )
{
    DestroyWindow( hwnd );
    return FALSE;
}

//get a surface pointer to our back buffer
ddscaps.dwCaps = DDSCAPS_BACKBUFFER;
ddreturn = lpPrimary->GetAttachedSurface(&ddscaps,
                                         &lpBackbuffer);

if( ddreturn != DD_OK )
{
    DestroyWindow ( hwnd );
    return FALSE;
}

// Create the offscreen surface
ddsd.dwFlags = DDSD_CAPS | DDSD_HEIGHT | DDSD_WIDTH;
ddsd.ddsCaps.dwCaps = DDSCAPS_OFFSCREENPLAIN;
ddsd.dwHeight = 64;
ddsd.dwWidth = 64;
ddreturn = lpDirectDrawObject->CreateSurface( &ddsd,
                                              &lpOffscreen, NULL );

if ( ddreturn != DD_OK )
{
    DestroyWindow( hwnd );
    return FALSE;
}

//copy a BMP file to the primary surface
if ( !BMPToDirectDrawSurface (lpPrimary, "display.bmp" ) )
{
    DestroyWindow ( hwnd );
    return FALSE;
}

//copy to the back buffer
if ( !BMPToDirectDrawSurface (lpBackbuffer, "display.bmp" ) )
{
    DestroyWindow ( hwnd );
    return FALSE;
}
```

```
    }

    //copy to the offscreen surface
    if ( !BMPToDirectDrawSurface (lpOffscreen, "sprite.bmp" ) )
    {
        DestroyWindow ( hwnd );
        return FALSE;
    }

    while ( 1 )
    {
        if( PeekMessage( &msg, NULL, 0, 0, PM_NOREMOVE ) )
        {
            if( !GetMessage( &msg, NULL, 0, 0 ) )
                return msg.wParam;
            TranslateMessage ( &msg );
            DispatchMessage ( &msg );
        }
        else if( !ActiveApp )
        {
            WaitMessage ();
        }
    }

} //WinMain

//*****
//WindowProc - receive and handle windows messages
//
//
//*****
long FAR PASCAL WindowProc( HWND hwnd, UINT message,
                           WPARAM wParam, LPARAM lParam )
{
    switch( message )
    {
        case WM_ACTIVATEAPP:
            ActiveApp = wParam;
            break;

        case WM_CREATE:
            break;

        case WM_KEYDOWN:
            switch ( wParam )
            {
                case VK_SPACE:
                    if ( !DrawFrame() )
                        DestroyWindow ( hwnd );
            }
    }
}
```

```
        break;
    case VK_ESCAPE:
        DestroyWindow ( hwnd );
        break;
    }
    break;

case WM_DESTROY:
    if ( lpDirectDrawObject != NULL )
    {
        if ( lpOffscreen != NULL )
            lpOffscreen->Release();
        if ( lpBackbuffer != NULL )
            lpBackbuffer->Release();
        if ( lpPrimary != NULL )
            lpPrimary->Release();
        lpDirectDrawObject->Release();
    }

    ShowCursor( TRUE );
    PostQuitMessage( 0 );
    break;

default:
    return DefWindowProc (hwnd,message,wParam,lParam);
}

return 0L;

} //WindowProc

//*****
//DrawFrame - Compose a frame of graphics, flip surfaces
//
//return:
//  BOOL TRUE - success
//*****
BOOL DrawFrame ( void )
{
    HRESULT ddreturn;
    static RECT rect = { 0, 0, 64, 64 };
    static DWORD XPos = 0, YPos = 0;
    static int XDelta = 2, YDelta = 2;

    //Blt the offscreen surface to the backbuffer
    do
    {
        ddreturn = lpBackbuffer->BltFast( XPos, YPos,
            lpOffscreen, &rect,
            DDBLTFAST_NOCOLORKEY|DDBLTFAST_WAIT);
    }
```

```
        if ( ddreturn == DDERR_SURFACELOST )
        {
            if ( !RestoreSurfaces() )
                return FALSE;
        }
    }while ( ddreturn != DD_OK );

    //Adjust destination coordinates
    if ( (XPos+XDelta) > 640-64 )
    {
        XDelta = -XDelta;
        XPos += XDelta;
    }
    if ( (YPos+YDelta) > 480-64 )
    {
        YDelta = -YDelta;
        YPos += YDelta;
    }
    //Let DirectDraw switch our surface pointers
    do
    {
        ddreturn = lpPrimary->Flip ( NULL, DDFLIP_WAIT );
        if ( ddreturn == DDERR_SURFACELOST )
        {
            if ( !RestoreSurfaces() )
                return FALSE;
        }
    }while ( ddreturn != DD_OK );

    return TRUE;

} //DrawFrame

//*****
//RestoreSurfaces - restores all lost surfaces
//
//returns:
//  BOOL True - success
//*****
BOOL RestoreSurfaces ( void )
{
    HRESULT ddreturn;

    ddreturn = lpPrimary->Restore();
    if ( ddreturn == DD_OK )
    {
        if ( !BMPToDirectDrawSurface (lpPrimary,"display.bmp" ))
            return FALSE;
    }
    else
    {

```

```

        return FALSE;
    }
    ddreturn = lpBackbuffer->Restore();
    if( ddreturn == DD_OK )
    {
        if (!BMPToDirectDrawSurface(lpBackbuffer,"display.bmp"))
            return FALSE;
    }
    else
    {
        return FALSE;
    }

    ddreturn = lpOffscreen->Restore();
    if( ddreturn == DD_OK )
    {
        if (!BMPToDirectDrawSurface(lpOffscreen,"sprite.bmp"))
            return FALSE;
    }
    else
    {
        return FALSE;
    }

    return TRUE;
}

//RestoreSurfaces

//*****
//BMPToDirectDrawSurface - opens and copies a BMP to a DD surface
//input:
// LPDIRECTDRAWSURFACE surface
// char *bmp - filename
//returns:
// BOOL TRUE - success
//*****
BOOL BMPToDirectDrawSurface ( LPDIRECTDRAWSURFACE surface,
                              char *bmp )
{
    HRESULT          ddreturn;
    DWORD            actualRead;
    HANDLE           hfile;
    char             path[MAX_PATH];
    BITMAPFILEHEADER BMPFileHead;
    BITMAPINFOHEADER BMPFileInfo;
    RGBQUAD          Palette[256];
    PALETTEENTRY      pe[256];
    LPDIRECTDRAWPALETTE lpDDPalette;
    DDSURFACEDESC     ddsd;
    int              i,src_width;
    BYTE             *lpDDMemory,*lpBMPMemory,*image;

    hfile = CreateFile( bmp, GENERIC_READ, FILE_SHARE_READ,

```

```
(LPSECURITY_ATTRIBUTES)NULL,  
OPEN_EXISTING, FILE_ATTRIBUTE_NORMAL,  
(HANDLE) NULL );  
  
if ( hfile == INVALID_HANDLE_VALUE )  
{  
    strcpy (path, "..\\");           //back up one directory  
    strcat( path, bmp );  
    hfile = CreateFile( path, GENERIC_READ, FILE_SHARE_READ,  
                        (LPSECURITY_ATTRIBUTES)NULL,  
                        OPEN_EXISTING, FILE_ATTRIBUTE_NORMAL,  
                        (HANDLE) NULL );  
    if ( hfile == INVALID_HANDLE_VALUE )  
        return FALSE;  
}  
  
// Read the BMP header and info structures  
if( !ReadFile ( hfile, &BMPFileHead, sizeof ( BMPFileHead ),  
               &actualRead, NULL) )  
    return FALSE;  
  
if( !ReadFile ( hfile, &BMPFileInfo, sizeof ( BMPFileInfo ),  
               &actualRead, NULL) )  
    return FALSE;  
  
//Make sure the BMP is 8-bit color  
if ( BMPFileInfo.biBitCount != 8 )  
    return FALSE;  
  
//get the BMP palette  
if ( !ReadFile ( hfile, Palette, sizeof( Palette ),  
               &actualRead, NULL ) )  
    return FALSE;  
  
//Allocate memory for image data  
image = (BYTE *) LocalAlloc ( LPTR,  
                              BMPFileInfo.biWidth *  
                              BMPFileInfo.biHeight );  
  
if( image == NULL )  
    return FALSE;  
  
//read BMP into memory  
if( !ReadFile ( hfile, image,  
               BMPFileInfo.biWidth * BMPFileInfo.biHeight,  
               &actualRead, NULL) )  
{  
    LocalFree( image );  
    return FALSE;  
}  
  
//copy BMP data to DirectDraw surface
```

```
memset ( &ddsd, 0, sizeof( DDSURFACEDESC ) );
ddsd.dwSize = sizeof( ddsd );
ddreturn = surface->Lock( NULL, &ddsd, 0, NULL );

if( ddreturn != DD_OK )
{
    LocalFree( image );
    return FALSE;
}

lpDDMemory = (BYTE *) ddsd.lpSurface;
lpBMPMemory = image+((BMPFileInfo.biHeight-1)*
                    BMPFileInfo.biWidth);

if ( BMPFileInfo.biWidth > ddsd.lPitch )
    src_width = ddsd.lPitch;
else
    src_width = BMPFileInfo.biWidth;

for( i = 0; i < BMPFileInfo.biHeight; i++ )
{
    memcpy ( lpDDMemory, lpBMPMemory, src_width );
    lpDDMemory += ddsd.lPitch;
    lpBMPMemory -= src_width;
}

surface->Unlock ( NULL );
LocalFree( image );
CloseHandle(hfile);

//create a DirectDraw palette for the surface
for( i=0; i<256; i++ )
{
    pe[i].peRed = Palette[i].rgbRed;
    pe[i].peGreen = Palette[i].rgbGreen;
    pe[i].peBlue = Palette[i].rgbBlue;
}
ddreturn = lpDirectDrawObject->CreatePalette( DDPCAPS_8BIT,
                                              pe,
                                              &lpDDPalette,
                                              NULL );

if ( ddreturn != DD_OK )
    return FALSE;
surface->SetPalette ( lpDDPalette );

return TRUE;
} //BMPToDirectDrawSurface
```


In the `WinMain` function, we perform mandatory DirectDraw initialization by establishing our DirectDraw object, setting the cooperation level, and choosing a display mode. We then create a page flippable primary surface with one back buffer and call **GetAttachedSurface** to retrieve an independent **DirectDrawSurface** object for the back buffer. At this point, we create the off-screen surface.

The **BMPToDirectDrawSurface** function is then used to fill all of the drawing surfaces with pixel data. The program in Chapter 6 seeded the two-page flipping surfaces with different BMP data to help illustrate the page flipping concept. In this program, we fill the display buffers with the same pixel data, flipping between them to create the illusion of one solid background on screen. If this concept remains unclear, try replacing the first two calls to **BMPToDirectDrawSurface** with the following lines:

```
if ( !BMPToDirectDrawSurface (lpPrimary,
    "c:\\newspeak\\chap05\\prim.bmp" ) )
if ( !BMPToDirectDrawSurface (lpBackbuffer,
    "c:\\newspeak\\chap05\\back.bmp" ) )
```

Although the color data in the PRIM.BMP and BACK.BMP files will not match the palette of the SPRITE.BMP file, you will be able to see that page flipping is indeed taking place.

Blitting of the off-screen surface takes place in the `DrawFrame` function when the **Spacebar** is pressed. The `DrawFrame` function first calls **BltFast** to copy the off-screen pixel data to the surface that is represented in **lpBackbuffer**. Is it possible that the blter chip will be busy with some other activity when we call **BltFast**. This is the reason we include the DO/WHILE loop, checking for a **DDERR_WASSTILLDRAWING** message from DirectDraw. Although this sort of checking is probably unnecessary in a program this simple, more complex DirectDraw applications will make larger demands of the blter chip and need to check for this condition.

After the call to **BltFast**, we update the simple X and Y positional values and check to make sure that the off-screen surface will not be blted to an illegal position on the next frame. It is important to remember that the **BltFast** function has no clipping abilities—clipping must be maintained by the programmer. A general protection fault will occur when **BltFast** receives source or destination coordinates that are outside of the surfaces legal boundaries. Once our coordinate checking is complete, we call the **Flip** function to switch the primary/back buffer object pointers.

USING OFF-SCREEN SURFACES AS BUFFERS

In the program given earlier, we are blting the pixel data in the off-screen surface destructively, that is, without saving and then later restoring the underlying pixels in our display buffers. While this type of code can achieve interesting results, it is usually more desirable to blt off-screen objects in a nondestructive fashion, giving the appearance of images moving smoothly over the background.

A number of techniques can be used to save or restore pixels that will be overwritten by a blt operation(s). The following OFFSCREEN1.CPP program uses two off-screen surfaces, one for each display buffer, to hold and then later restore underlying pixel data using **BltFast**.

THE OFFSCREEN1.CPP PROGRAM

```
//*****
//DirectDraw Programming
//Bret Timmins
//
//(C)1995 M&T Books
//
//off_scr1.cpp
// Demonstrates blting an offscreen surface in a
// page flipping enviroment. Adds buffering for clean
// animation of the offscreen surface
//*****
#define WIN32_LEAN_AND_MEAN
#include <windows.h>
#include <windowsx.h>
#ifdef __BORLANDC__ && defined( __WIN32__ )
#define _WIN32
#endif
#include <ddraw.h>
#include <stdio.h>
#include <stdlib.h>
#include <stdarg.h>

#define EVEN 0
#define ODD 1
#define NEG 0xffffffff

LPDIRECTDRAW      lpDirectDrawObject; //DD object
LPDIRECTDRAWSURFACE lpPrimary;        //DD primary surface
LPDIRECTDRAWSURFACE lpBackbuffer;     //DD back buffer surface
```

```

LPDIRECTDRAW_SURFACE lpOffscreen;           //Offscreen surface
LPDIRECTDRAW_SURFACE lpOffBufferEven;        //Buffers pixel data
                                              //that would be destroyed
                                              //by BltFast for the
                                              //primary surface
LPDIRECTDRAW_SURFACE lpOffBufferOdd;         //Buffers pixel data
                                              //that would be destroyed
                                              //by BltFast for the
                                              //back buffer surface
BOOL                                     ActiveApp;    //Is this program active?

int PASCAL WinMain( HINSTANCE hInstance, HINSTANCE hPrevInstance,
                    LPSTR lpCmdLine, int nCmdShow);
long FAR PASCAL WindowProc ( HWND hwnd, UINT message,
                             WPARAM wParam, LPARAM lParam );

BOOL DrawFrame ( void );
BOOL RestoreSurfaces ( void );
BOOL BMPToDirectDrawSurface ( LPDIRECTDRAW_SURFACE surface,
                              char *bmp );

//*****
//WinMain - mandatory windows init function
//
//*****
int PASCAL WinMain( HINSTANCE hInstance, HINSTANCE hPrevInstance,
                    LPSTR lpCmdLine, int nCmdShow)
{
    MSG          msg;
    HWND         hwnd;
    WNDCLASS     wc;
    static char  ClassName[] = "Offscreen1";
    DDSURFACEDESC ddsd;
    DDSCAPS      ddscaps;
    HRESULT      ddreturn;

    lpCmdLine = lpCmdLine;
    hPrevInstance = hPrevInstance;

    //register and realize our display window
    wc.style = CS_HREDRAW | CS_VREDRAW;
    wc.lpfnWndProc = WindowProc;
    wc.cbClsExtra = 0;
    wc.cbWndExtra = 0;
    wc.hInstance = hInstance;
    wc.hIcon = LoadIcon( hInstance, IDI_APPLICATION );
    wc.hCursor = LoadCursor( NULL, IDC_ARROW );
    wc.hbrBackground = NULL;
    wc.lpszMenuName = ClassName;
    wc.lpszClassName = ClassName;
    RegisterClass( &wc );

```

```
//create a full screen window so that GDI won't ever be
//called
hwnd = CreateWindowEx(
    WS_EX_TOPMOST,
    ClassName,
    ClassName,
    WS_POPUP,
    0,
    0,
    GetSystemMetrics(SM_CXSCREEN),
    GetSystemMetrics(SM_CYSCREEN),
    NULL,
    NULL,
    hInstance,
    NULL );

if( !hwnd )
    return FALSE;

ShowWindow( hwnd, nCmdShow );
UpdateWindow( hwnd );
SetFocus( hwnd );
ShowCursor( FALSE );

//Instanciate our DirectDraw object

ddreturn = DirectDrawCreate( NULL,
                             &lpDirectDrawObject, NULL );
if ( ddreturn != DD_OK )
{
    DestroyWindow( hwnd );
    return FALSE;
}

ddreturn = lpDirectDrawObject->SetCooperativeLevel( hwnd,
                                                     DDSCL_EXCLUSIVE | DDSCL_FULLSCREEN );
if ( ddreturn != DD_OK )
{
    DestroyWindow( hwnd );
    return FALSE;
}

// Set the video mode to 640x480x8
ddreturn = lpDirectDrawObject->SetDisplayMode( 640, 480, 8);
if ( ddreturn != DD_OK )
{
    DestroyWindow( hwnd );
    return FALSE;
}
```

```
// Create the primary surface and one back buffer surface
ddsd.dwSize = sizeof( ddsd );
ddsd.dwFlags = DDSF_CAPS | DDSF_BACKBUFFERCOUNT;
ddsd.ddsCaps.dwCaps = DDSCAPS_PRIMARYSURFACE |
                    DDSCAPS_FLIP |
                    DDSCAPS_COMPLEX;
ddsd.dwBackBufferCount = 1;
ddreturn = lpDirectDrawObject->CreateSurface( &ddsd,
                                              &lpPrimary, NULL );

if ( ddreturn != DD_OK )
{
    DestroyWindow( hwnd );
    return FALSE;
}

//get a surface pointer to our back buffer
ddscaps.dwCaps = DDSCAPS_BACKBUFFER;
ddreturn = lpPrimary->GetAttachedSurface ( &ddscaps,
                                           &lpBackbuffer);

if( ddreturn != DD_OK )
{
    DestroyWindow ( hwnd );
    return FALSE;
}

// Create the main offscreen surface
ddsd.dwFlags = DDSF_CAPS | DDSF_HEIGHT | DDSF_WIDTH;
ddsd.ddsCaps.dwCaps = DDSCAPS_OFFSCREENPLAIN;
ddsd.dwHeight = 64;
ddsd.dwWidth = 64;
ddreturn = lpDirectDrawObject->CreateSurface ( &ddsd,
                                              &lpOffscreen,
                                              NULL );

if ( ddreturn != DD_OK )
{
    DestroyWindow( hwnd );
    return FALSE;
}

// Create the offscreen surface buffer EVEN
ddsd.dwFlags = DDSF_CAPS | DDSF_HEIGHT | DDSF_WIDTH;
ddsd.ddsCaps.dwCaps = DDSCAPS_OFFSCREENPLAIN;
ddsd.dwHeight = 64;
ddsd.dwWidth = 64;
ddreturn = lpDirectDrawObject->CreateSurface ( &ddsd,
                                              &lpOffBufferEven,
                                              NULL );

if ( ddreturn != DD_OK )
{
```

```
DestroyWindow( hwnd );
return FALSE;
}

// Create the offscreen surface buffer ODD
ddsd.dwFlags = DDSD_CAPS | DDSD_HEIGHT | DDSD_WIDTH;
ddsd.ddsCaps.dwCaps = DDSCAPS_OFFSCREENPLAIN;
ddsd.dwHeight = 64;
ddsd.dwWidth = 64;
ddreturn = lpDirectDrawObject->CreateSurface ( &ddsd,
                                                &lpOffBufferOdd,
                                                NULL );

if ( ddreturn != DD_OK )
{
    DestroyWindow ( hwnd );
    return FALSE;
}

//copy a BMP file to the primary surface
if ( !BMPToDirectDrawSurface (lpPrimary, "display.bmp" ) )
{
    DestroyWindow ( hwnd );
    return FALSE;
}

//copy to the back buffer
if ( !BMPToDirectDrawSurface (lpBackbuffer, "display.bmp" ) )
{
    DestroyWindow ( hwnd );
    return FALSE;
}

//copy to the offscreen surface
if ( !BMPToDirectDrawSurface (lpOffscreen, "sprite.bmp" ) )
{
    DestroyWindow ( hwnd );
    return FALSE;
}

while ( 1 )
{
    if( PeekMessage( &msg, NULL, 0, 0, PM_NOREMOVE ) )
    {
        if( !GetMessage( &msg, NULL, 0, 0 ) )
            return msg.wParam;
        TranslateMessage ( &msg );
        DispatchMessage ( &msg );
    }
    else if( !ActiveApp )
    {

```

```
        WaitMessage ();
    }
}

} //WinMain

//*****
//WindowProc - receive and handle windows messages
//
//
//*****
long FAR PASCAL WindowProc( HWND hwnd, UINT message,
                           WPARAM wParam, LPARAM lParam )
{
    switch( message )
    {
        case WM_ACTIVATEAPP:
            ActiveApp = wParam;
            break;

        case WM_CREATE:
            break;

        case WM_KEYDOWN:
            switch ( wParam )
            {
                case VK_SPACE:
                    if ( !DrawFrame() )
                        DestroyWindow ( hwnd );
                    break;
                case VK_ESCAPE:
                    DestroyWindow ( hwnd );
                    break;
            }
            break;

        case WM_DESTROY:
            if ( lpDirectDrawObject != NULL )
            {
                if ( lpOffBufferEven != NULL )
                    lpOffBufferEven->Release();
                if ( lpOffBufferOdd != NULL )
                    lpOffBufferOdd->Release();
                if ( lpOffscreen != NULL )
                    lpOffscreen->Release();
                if ( lpBackbuffer != NULL )
                    lpBackbuffer->Release();
                if ( lpPrimary != NULL )
                    lpPrimary->Release();
                lpDirectDrawObject->Release();
            }
    }
}
```

```

    }

    ShowCursor ( TRUE );
    PostQuitMessage ( 0 );
    break;

default:
    return DefWindowProc (hwnd,message,wParam,lParam);
}

return 0L;

} // WindowProc

//*****
// DrawFrame - Compose a frame of graphics, flip surfaces
//
// return:
//  BOOL TRUE - success
//*****
BOOL DrawFrame ( void )
{
    HRESULT ddreturn;
    static RECT rect = { 0, 0, 64, 64 };
    RECT BufferRect;
    static DWORD XPos = 0, YPos = 0;
    static DWORD XOld[2] = { NEG, NEG };
    static DWORD YOld[2] = { NEG, NEG };
    static int XDelta = 2, YDelta = 2;
    static BOOL Frame = EVEN;
    static LPDIRECTDRAWSURFACE lpBuffer;

    if ( Frame == EVEN )
        lpBuffer = lpOffBufferEven;
    else
        lpBuffer = lpOffBufferOdd;

    // Restore back buffer pixels from previous frame
    if ( XOld[Frame] != NEG )
    {
        do
        {
            ddreturn = lpBackbuffer->BltFast( XOld[Frame],
                                                YOld[Frame],
                                                lpBuffer, &rect,
                                                DDBLTFAST_NOCOLORKEY|DDBLTFAST_WAIT);
            if ( ddreturn == DDERR_SURFACELOST )
            {
                if ( !RestoreSurfaces() )
                    return FALSE;
                // case - after restore, buffers are invalid,
            }
        } while ( ddreturn == DDERR_SURFACELOST );
    }
}

```



```
        //rebuild them
        XOld[0] = NEG; XOld[1] = NEG;
        YOld[0] = NEG; YOld[1] = NEG;
        break;
    }
}while ( ddreturn != DD_OK );
}

//Save back buffer pixels to the offscreen buffer
BufferRect.left = XPos;
BufferRect.top = YPos;
BufferRect.right = XPos+64;
BufferRect.bottom = YPos+64;

do
{
    ddreturn = lpBuffer->BltFast( 0, 0,
                                lpBackbuffer, &BufferRect,
                                DDBLTFAST_NOCOLORKEY|DDBLTFAST_WAIT );
    if ( ddreturn == DDERR_SURFACELOST )
    {
        if ( !RestoreSurfaces() )
            return FALSE;
    }
}while ( ddreturn != DD_OK );

//Blt the offscreen surface to the back buffer
do
{
    ddreturn = lpBackbuffer->BltFast( XPos, YPos,
                                lpOffscreen, &rect,
                                DDBLTFAST_NOCOLORKEY|DDBLTFAST_WAIT );
    if ( ddreturn == DDERR_SURFACELOST )
    {
        if ( !RestoreSurfaces() )
            return FALSE;
    }
}while ( ddreturn != DD_OK );

//Save old positions for restoring on the next frame
XOld[Frame] = XPos;
YOld[Frame] = YPos;

//set of appropriate buffers for the next frame
if ( Frame == EVEN )
    Frame = ODD;
else
    Frame = EVEN;

//Adjust destination coordinates for next frame
if ( (XPos+XDelta) > 640-64 )
```

```

{
    XDelta = -XDelta;
    XPos += XDelta;
}
if ( (YPos+YDelta) > 480-64 )
{
    YDelta = -YDelta;
    YPos += YDelta;
}

//Let DirectDraw switch our surface pointers
do
{
    ddreturn = lpPrimary->Flip ( NULL, DDFLIP_WAIT );
    if ( ddreturn == DDERR_SURFACELOST )
    {
        if ( !RestoreSurfaces() )
            return FALSE;
    }
}while ( ddreturn != DD_OK );

return TRUE;

} //DrawFrame

//*****
//RestoreSurfaces - restores all lost surfaces
//
//returns:
//  BOOL True - success
//*****
BOOL RestoreSurfaces ( void )
{
    HRESULT ddreturn;

    ddreturn = lpPrimary->Restore();
    if ( ddreturn == DD_OK )
    {
        if ( !BMPToDirectDrawSurface (lpPrimary, "display.bmp" ) )
            return FALSE;
    }
    else
    {
        return FALSE;
    }
    ddreturn = lpBackbuffer->Restore();
    if ( ddreturn == DD_OK )
    {
        if ( !BMPToDirectDrawSurface(lpBackbuffer,"display.bmp") )
            return FALSE;
    }
    else

```

```

    {
        return FALSE;
    }

    ddreturn = lpOffscreen->Restore();
    if( ddreturn == DD_OK )
    {
        if (!BMPToDirectDrawSurface(lpOffscreen,"sprite.bmp"))
            return FALSE;
    }else
    {
        return FALSE;
    }

    ddreturn = lpOffBufferEven->Restore();
    if( ddreturn != DD_OK )
        return FALSE;

    ddreturn = lpOffBufferOdd->Restore();
    if( ddreturn != DD_OK )
        return FALSE;

    return TRUE;

} //RestoreSurfaces

//*****
//BMPToDirectDrawSurface - opens and copies a BMP to a DD surface
//input:
// LPDIRECTDRAWSURFACE surface
// char *bmp - filename
//returns:
// BOOL TRUE - success
//*****
BOOL BMPToDirectDrawSurface ( LPDIRECTDRAWSURFACE surface,
                               char *bmp )
{
    HRESULT          ddreturn;
    DWORD            actualRead;
    HANDLE           hfile;
    char             path[MAX_PATH];
    BITMAPFILEHEADER BMPFileHead;
    BITMAPINFOHEADER BMPFileInfo;
    RGBQUAD          Palette[256];
    PALETTEENTRY      pe[256];
    LPDIRECTDRAWPALETTE lpDDPalette;
    DDSURFACEDESC    ddsd;
    int              i,src_width;
    BYTE             *lpDDMemory,*lpBMPMemory,*image;

```

```
hfile = CreateFile( bmp, GENERIC_READ, FILE_SHARE_READ,
                   (LPSECURITY_ATTRIBUTES)NULL,
                   OPEN_EXISTING, FILE_ATTRIBUTE_NORMAL,
                   (HANDLE) NULL );

if ( hfile == INVALID_HANDLE_VALUE )
{
    strcpy (path, "..\\");           //back up one directory
    strcat( path, bmp );
    hfile = CreateFile( path, GENERIC_READ, FILE_SHARE_READ,
                       (LPSECURITY_ATTRIBUTES)NULL,
                       OPEN_EXISTING, FILE_ATTRIBUTE_NORMAL,
                       (HANDLE) NULL );
    if ( hfile == INVALID_HANDLE_VALUE )
        return FALSE;
}

// Read the BMP header and info structures
if( !ReadFile ( hfile, &BMPFileHead, sizeof ( BMPFileHead ),
               &actualRead, NULL) )
    return FALSE;

if( !ReadFile ( hfile, &BMPFileInfo, sizeof ( BMPFileInfo ),
               &actualRead, NULL) )
    return FALSE;

//Make sure the BMP is 8-bit color
if ( BMPFileInfo.biBitCount != 8 )
    return FALSE;

//get the BMP palette
if ( !ReadFile ( hfile, Palette, sizeof( Palette ),
               &actualRead, NULL ) )
    return FALSE;

//Allocate memory for image data
image = (BYTE *) LocalAlloc ( LPTR,
                              BMPFileInfo.biWidth *
                              BMPFileInfo.biHeight );

if( image == NULL )
    return FALSE;

//read BMP into memory
if( !ReadFile ( hfile, image,
               BMPFileInfo.biWidth * BMPFileInfo.biHeight,
               &actualRead, NULL) )
{
    LocalFree( image );
    return FALSE;
}
```

```
//copy BMP data to DirectDraw surface
memset ( &ddsd, 0, sizeof( DDSURFACEDESC ) );
ddsd.dwSize = sizeof( ddsd );
ddreturn = surface->Lock( NULL, &ddsd, 0, NULL );

if( ddreturn != DD_OK )
{
    LocalFree( image );
    return FALSE;
}

lpDDMemory = (BYTE *) ddsd.lpSurface;
lpBMPMemory = image+((BMPFileInfo.biHeight-1)*
                    BMPFileInfo.biWidth);

if ( BMPFileInfo.biWidth > ddsd.lPitch )
    src_width = ddsd.lPitch;
else
    src_width = BMPFileInfo.biWidth;

for( i = 0; i < BMPFileInfo.biHeight; i++ )
{
    memcpy ( lpDDMemory, lpBMPMemory, src_width );
    lpDDMemory += ddsd.lPitch;
    lpBMPMemory -= src_width;
}

surface->Unlock ( NULL );
LocalFree( image );
CloseHandle(hfile);

//create a DirectDraw palette for the surface
for( i=0; i<256; i++ )
{
    pe[i].peRed = Palette[i].rgbRed;
    pe[i].peGreen = Palette[i].rgbGreen;
    pe[i].peBlue = Palette[i].rgbBlue;
}
ddreturn = lpDirectDrawObject->CreatePalette( DDPCAPS_8BIT,
                                              pe,
                                              &lpDDPalette,
                                              NULL );

if ( ddreturn != DD_OK )
    return FALSE;
surface->SetPalette ( lpDDPalette );

return TRUE;
} //BMPToDirectDrawSurface
```

In the **WinMain** function, we create two additional off-screen surfaces, **lpOffscreenEven** and **lpOffscreenOdd**. Since these surfaces will hold pixels from the display buffers, there is no need to prefill them with any pixel data.

The first act of the **DrawFrame** function is to restore the pixel data that was saved on the previous frame. Conceptually, this is a simple matter, but it is complicated by the fact that we don't want this section of code to executed until both of the off-screen surfaces have been filled with pixel data. On the third iteration of **DrawFrame**, this section of code will execute as we begin restoring the pixels that were overwritten in the display buffers.

The second section of **DrawFrame** uses **BltFast** to save the pixels in the current back buffer before writing off-screen image to the back-buffer surface. Note the slightly different usage of the **RECT** structure within this part of the function.

After the off-screen image has been blted, we save the current X,Y position of the off-screen surface in the **Xold** and **Yold DWORD** arrays. We then toggle the static **Frame** variable, ensuring that on the next iteration of **DrawFrame**, we will be restoring data to the correct display surface.

Blting: Transparent Blting

In Chapter 7, we used the **BltFast** function to blt a rectangular off-screen image to a display surface. This type of straightforward blting definitely has its uses, but if we want to blt a circular shape nondestructively over the display surface, for example, we must use a transparent blt.

In this chapter, we will see how **DirectDraw** implements this type of blting by writing a program that defines a **DirectDraw** color key and uses **BltFast**'s transparent blting option.

THE TRANSPARENT PIXEL

Fundamental to game programming, a transparent blt operation can be thought of as a simple pixel filter. To blt nonrectangular images to the display surface, we must ensure that unwanted pixels from the source image aren't copied to the display surface. In the simplest case, this means designating one pixel value within the off-screen image as the nondisplayable, or transparent, pixel. When this pixel value is encountered during the blt operation, it will not be copied to the display surface, and the user will see a nonrectangular image blted over the display background.

Figure 8.1 illustrates a simple transparent blt.

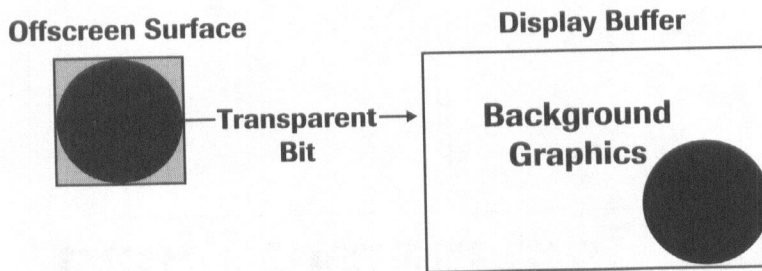


Figure 8.1 A Simple transparent blt.

USING TRANSPARENCY FOR DEPTH EFFECTS

In the previous example, the only pixels that are relevant for the transparent blt operation are contained in the off-screen image. The pixels on the display surface (or the destination surface) are either preserved or overwritten, depending on the type of pixel, either transparent or nontransparent, that is to be blted at that moment. We will refer to this type of transparent blting as being keyed from the source image, or *source keying*.

DirectDraw also allows transparent blts to be keyed from the destination surface. In this type of blting, we define a pixel value, or a range of pixel values, within the destination surface that can be replaced with pixels from the source image. Unlike source keying, where the off-screen image always appears on top of the display surface, *destination keying* allows the source image to appear behind other elements within the background, producing the illusion of depth.

THE COLOR KEY

To use transparent blting under DirectDraw, we must define a color key. A DirectDraw color key defines which pixels within a given drawing surface will be transparent. A color key is a pixel value, or in the case of 8-bit video modes, a palette index value. For 8-bit video modes, the possible color key range is 0-255. For 16- and 24-bit video modes, the color key(s) are actual color intensity values. With such a large number of possible colors in these higher color modes, it is more useful to express a color key in terms of an RGB value. Instead of specifying

the pure white color as 16777216, we can take advantage of a Windows macro and write the number as RGBQUAD (255,255,255). DirectDraw and color usage will be fully covered in Chapter 10: DirectDraw Color.

THE DDColorKEY STRUCTURE

A color key is specified by filling in the two fields of the **DDColorKEY** structure:

```
DWORD dwColorSpaceLowValue
DWORD dwColorSpaceHighValue
```

For a single-pixel color key, the **dwColorSpaceLowValue** and **dwColorSpaceHighValue** are the same. That is, if the video mode is 8 bit and the desired color key is 10, the **DDColorKEY** structure would equal:

```
dwColorSpaceLow = 10;
dwColorSpaceHigh = 10;
```

A range of color keys can also be used (Microsoft refers to this as a *color space*). For 8-bit graphics, this range defines a beginning and ending set of palette indexes that will be color keys. In higher color modes, the low and high fields define a set of color intensities that will be considered color keys.



NOTE

At this time, most blter chips can't support color key ranges. For optimum performance with most video cards, one color key per surface is recommended.

THE SetColorKey FUNCTION

The **SetColorKey** member function applies a **DDColorKEY** structure to a drawing surface:

```
HRESULT SetColorKey ( LPDIRECTDRAWSURFACE lpDDSurface,
                     DWORD dwFlags,
                     LPDDColorKEY lpDDColorKey)
```

lpDDSurface points to the drawing surface that will receive the color key. The **dwFlags** parameter determines the type of color key that will be applied to the surface:

DDCKEY_COLORSPACE

This equate informs DirectDraw that a range of color keys are requested.

DDCKEY_DESTBLT

Tells DirectDraw to apply the color key to the destination surface.

DDCKEY_DESTOVERLAY

Tells DirectDraw to apply the color key to the destination surface. This equate is only valid when applied to an overlay surface.

DDCKEY_SRCBLT

Tells DirectDraw to apply the color key to the source surface.

DDCKEY_SRCOVERLAY

Tells DirectDraw to apply the color key to the source surface. This equate is only valid when applied to an overlay surface.

lpDDColorKey points to a **DDCOLORKEY** structure. A return of **DD_OK** indicates success. An error return usually indicates that the blter hardware is not capable of using color keys. The transparent blting capabilities of the hardware can be determined by calling the **GetCaps** function and checking the **dwCKKeyCaps** field of the **DDCAPS** structure.

THE GETCOLORKEY FUNCTION

This function returns the color key or range of color keys, applied to a given surface:

```
HRESULT GetColorKey ( LPDIRECTDRAW_SURFACE lpDDSurface,  
                     DWORD dwFlags,  
                     LPDDCOLORKEY lpDDColorKey)
```

lpDDSurface points to the drawing surface. Since it is possible to have multiple color key types applied to a surface, the **dwFlags** parameter is used to specify the

type of color key that should be returned. The usage of this parameter is identical to **dwFlags** in the **SetColorKey** function. **lpDDColorKey** points to a **DDCOLORKEY** structure that will be filled with the color key information for this surface. A return value of **DD_OK** indicates that the surface has an attached color key of the type specified in **dwFlags**, and the structure pointed at by **lpDDColorKey** contains valid color key information. A return of **DDERR_NOCOLORKEY** indicates that the surface doesn't have a color key of the given type.

THE TB_SRC.CPP PROGRAM

```
//*****
//DirectDraw Programming
//Bret M. Timmins
//
//(C)1995 M&T Books
//
//tb_src.cpp
// Demonstrates transparent blting using blt fast
// with a source color key
//*****

#define WIN32_LEAN_AND_MEAN
#include <windows.h>
#include <windowsx.h>

#if defined( __BORLANDC__ ) && defined( __WIN32__ )
#define _WIN32
#endif

#include "gdk\sdk\inc\ddraw.h"
#include <stdio.h>
#include <stdlib.h>
#include <stdarg.h>

#define EVEN 0
#define ODD 1
#define NEG 0xffffffff

LPDIRECTDRAW          lpDirectDrawObject;    // DirectDraw object
LPDIRECTDRAWSURFACE   lpPrimary;             // DirectDraw primary
surface
LPDIRECTDRAWSURFACE   lpBackbuffer;          // DirectDraw primary surface
LPDIRECTDRAWSURFACE   lpOffscreen;           // Offscreen surface
LPDIRECTDRAWSURFACE   lpOffBufferEven;       // Offscreen surface,
```

```

buffers
// pixel data that would be
// destroyed by BltFast for
// the primary surface
LPDIRECTDRAWSURFACE    lpOffBufferOdd;    // Offscreen surface,
buffers
// pixel data that would be
// destroyed by BltFast for
// the back buffer surface
BOOL                    ActiveApp;        // Is this program active?

int PASCAL WinMain( HINSTANCE hInstance, HINSTANCE hPrevInstance,
                    LPSTR lpCmdLine, int nCmdShow);
long FAR PASCAL WindowProc ( HWND hwnd, UINT message,
                             WPARAM wParam, LPARAM lParam );

BOOL DrawFrame ( void );
BOOL RestoreSurfaces ( void );
BOOL BMPToDirectDrawSurface( LPDIRECTDRAWSURFACE surface, char *bmp );

//*****
//WinMain - mandatory windows init function
//
//*****
int PASCAL WinMain( HINSTANCE hInstance, HINSTANCE hPrevInstance,
                    LPSTR lpCmdLine, int nCmdShow)
{
    MSG        msg;
    HWND        hwnd;
    WNDCLASS    wc;
    static char ClassName[] = "SRC_BLT";
    DDSURFACEDESC ddsd;
    DDSCAPS     ddscaps;
    DDCOLORKEY   ddck;
    HRESULT      ddreturn;

    lpCmdLine = lpCmdLine;
    hPrevInstance = hPrevInstance;

    //register and realize our display window
    wc.style = CS_HREDRAW | CS_VREDRAW;
    wc.lpfnWndProc = WindowProc;
    wc.cbClsExtra = 0;
    wc.cbWndExtra = 0;
    wc.hInstance = hInstance;
    wc.hIcon = LoadIcon( hInstance, IDI_APPLICATION );
    wc.hCursor = LoadCursor( NULL, IDC_ARROW );
    wc.hbrBackground = NULL;
    wc.lpszMenuName = ClassName;
    wc.lpszClassName = ClassName;
    RegisterClass( &wc );

```

```
//create a full screen window so that GDI won't ever be called
hwnd = CreateWindowEx(
    WS_EX_TOPMOST,
    ClassName,
    ClassName,
    WS_POPUP,
    0,
    0,
    GetSystemMetrics(SM_CXSCREEN),
    GetSystemMetrics(SM_CYSCREEN),
    NULL,
    NULL,
    hInstance,
    NULL );

if( !hwnd )
    return FALSE;

ShowWindow( hwnd, nCmdShow );
UpdateWindow( hwnd );
SetFocus( hwnd );
ShowCursor( FALSE );

//Instantiate our DirectDraw object
ddreturn = DirectDrawCreate( NULL,
                             &lpDirectDrawObject, NULL );
if ( ddreturn != DD_OK )
{
    DestroyWindow( hwnd );
    return FALSE;
}

ddreturn = lpDirectDrawObject->SetCooperativeLevel( hwnd,
    DDSCL_EXCLUSIVE | DDSCL_FULLSCREEN );
if ( ddreturn != DD_OK )
{
    DestroyWindow( hwnd );
    return FALSE;
}

// Set the video mode to 640x480x8
ddreturn = lpDirectDrawObject->SetDisplayMode( 640, 480, 8);
if ( ddreturn != DD_OK )
{
    DestroyWindow( hwnd );
    return FALSE;
}

// Create the primary surface and one back buffer surface
```

```

ddsd.dwSize = sizeof( ddsd );
ddsd.dwFlags = DDSD_CAPS | DDSD_BACKBUFFERCOUNT;
ddsd.ddsCaps.dwCaps = DDSCAPS_PRIMARYSURFACE |
                    DDSCAPS_FLIP |
                    DDSCAPS_COMPLEX;

ddsd.dwBackBufferCount = 1;
ddreturn = lpDirectDrawObject->CreateSurface( &ddsd, &lpPrimary, NULL
);
if ( ddreturn != DD_OK )
{
    DestroyWindow( hwnd );
    return FALSE;
}

//get a surface pointer to our back buffer
ddscaps.dwCaps = DDSCAPS_BACKBUFFER;
ddreturn = lpPrimary->GetAttachedSurface ( &ddscaps, &lpBackbuffer);
if( ddreturn != DD_OK )
{
    DestroyWindow ( hwnd );
    return FALSE;
}

// Create the main offscreen surface
ddsd.dwFlags = DDSD_CAPS | DDSD_HEIGHT | DDSD_WIDTH;
ddsd.ddsCaps.dwCaps = DDSCAPS_OFFSCREENPLAIN;
ddsd.dwHeight = 480;
ddsd.dwWidth = 512;
ddreturn = lpDirectDrawObject->CreateSurface ( &ddsd,
                                                &lpOffscreen, NULL );

if ( ddreturn != DD_OK )
{
    DestroyWindow( hwnd );
    return FALSE;
}

// Create the offscreen surface buffer EVEN
ddsd.dwFlags = DDSD_CAPS | DDSD_HEIGHT | DDSD_WIDTH;
ddsd.ddsCaps.dwCaps = DDSCAPS_OFFSCREENPLAIN;
ddsd.dwHeight = 120;
ddsd.dwWidth = 128;
ddreturn = lpDirectDrawObject->CreateSurface ( &ddsd,
                                                &lpOffBufferEven, NULL );

if ( ddreturn != DD_OK )
{
    DestroyWindow( hwnd );
    return FALSE;
}

// Create the offscreen surface buffer ODD

```

```

ddsd.dwFlags = DDSD_CAPS | DDSD_HEIGHT | DDSD_WIDTH;
ddsd.ddsCaps.dwCaps = DDSCAPS_OFFSCREENPLAIN;
ddsd.dwHeight = 120;
ddsd.dwWidth = 128;
ddreturn = lpDirectDrawObject->CreateSurface ( &ddsd,
                                                &lpOffBufferOdd, NULL );
    if ( ddreturn != DD_OK )
    {
        DestroyWindow ( hwnd );
        return FALSE;
    }

//Set the transparent color key for the off screen surface
ddck.dwColorSpaceLowValue = 255;
ddck.dwColorSpaceHighValue = 255;
lpOffscreen->SetColorKey( DDCKEY_SRCBLT, &ddck );

//copy a BMP file to the primary surface
if ( !BMPToDirectDrawSurface (lpPrimary, "stars.bmp" ) )
{
    DestroyWindow ( hwnd );
    return FALSE;
}

//copy to the back buffer
if ( !BMPToDirectDrawSurface (lpBackbuffer, "stars.bmp" ) )
{
    DestroyWindow ( hwnd );
    return FALSE;
}

//copy to the offscreen surface
if ( !BMPToDirectDrawSurface (lpOffscreen, "planet.bmp" ) )
{
    DestroyWindow ( hwnd );
    return FALSE;
}

while( 1 )
{
    if( PeekMessage( &msg, NULL, 0, 0, PM_NOREMOVE ) )
    {
        if( !GetMessage( &msg, NULL, 0, 0 ) )
            return msg.wParam;
        TranslateMessage(&msg);
        DispatchMessage(&msg);
    }else if ( ActiveApp )
    {
        DrawFrame ();
    }else

```

```

    {
        WaitMessage ();
    }
}

} // WinMain

// *****
// WindowProc - receive and handle windows messages
//
// *****
long FAR PASCAL WindowProc( HWND hwnd, UINT message,
                           WPARAM wParam, LPARAM lParam )
{
    switch( message )
    {
        case WM_ACTIVATEAPP:
            ActiveApp = wParam;
            break;

        case WM_CREATE:
            break;

        case WM_KEYDOWN:
            switch ( wParam )
            {
                case VK_ESCAPE:
                    DestroyWindow ( hwnd );
                    break;
            }
            break;

        case WM_DESTROY:
            if ( lpDirectDrawObject != NULL )
            {
                if ( lpOffBufferEven != NULL )
                    lpOffBufferEven->Release();
                if ( lpOffBufferOdd != NULL )
                    lpOffBufferOdd->Release();
                if ( lpOffscreen != NULL )
                    lpOffscreen->Release();
                if ( lpPrimary != NULL )
                    lpPrimary->Release();
                lpDirectDrawObject->Release();
            }

            ShowCursor ( TRUE );
            PostQuitMessage ( wParam );
            break;
    }
}

```



```
default:
    return DefWindowProc ( hwnd, message, wParam, lParam );
}

return 0L;

} // WindowProc

// *****
// DrawFrame - Compose a frame of graphics, flip surfaces
//
// return:
//   BOOL TRUE - success
// *****
BOOL DrawFrame ( void )
{
    HRESULT ddreturn;
    static RECT rect = { 0, 0, 128, 120 };
    RECT BufferRect, AnimRect;
    static DWORD XPos = 100, YPos = 180;
    static DWORD XOld[2] = { NEG, NEG };
    static DWORD YOld[2] = { NEG, NEG };
    static BOOL Frame = EVEN;
    static LPDIRECTDRAWSURFACE lpBuffer;
    static LONG XStart [16] = { 0, 128, 256, 384, 0, 128, 256, 384,
                                0, 128, 256, 384, 0, 128, 256, 384 };
};

    static LONG YStart [16] = { 0, 0, 0, 0, 120, 120, 120, 120,
                                240, 240, 240, 240, 360, 360, 360, 360 };
    static int AnimFrame = 0;
    static DWORD TickCount = GetTickCount();

    // simple timing
    if ( (GetTickCount() - TickCount) < 50 )
        return TRUE;
    TickCount = GetTickCount();

    if ( Frame == EVEN )
        lpBuffer = lpOffBufferEven;
    else
        lpBuffer = lpOffBufferOdd;

    // Restore back buffer pixels from previous frame
    if ( XOld[Frame] != NEG )
    {
        do
        {
            ddreturn = lpBackbuffer->BltFast( XOld[Frame], YOld[Frame],
                                                lpBuffer, &rect,
                                                DDBLTFAST_NOCOLORKEY | DDBLTFAST_WAIT );
        }
    }
}
```

```

        if ( ddreturn == DDERR_SURFACELOST )
        {
            if ( !RestoreSurfaces() )
                return FALSE;
            //case-after restore, buffers are invalid, rebuild
them
            X01d[0] = NEG; X01d[1] = NEG;
            Y01d[0] = NEG; Y01d[1] = NEG;
            break;
        }
    }while ( ddreturn != DD_OK );
}

//Save back buffer pixels to the offscreen buffer
BufferRect.left = XPos;
BufferRect.top = YPos;
BufferRect.right = XPos+128;
BufferRect.bottom = YPos+120;
do
{
    ddreturn = lpBuffer->BltFast( 0, 0,
                                lpBackbuffer, &BufferRect,
                                DDBLTFAST_NOCOLORKEY|DDBLTFAST_WAIT );
    if ( ddreturn == DDERR_SURFACELOST )
    {
        if ( !RestoreSurfaces() )
            return FALSE;
    }
}while ( ddreturn != DD_OK );

//Blt new a animation frame from the off screen surface
AnimRect.left = XStart[AnimFrame];
AnimRect.top = YStart[AnimFrame];
AnimRect.right = AnimRect.left+128;
AnimRect.bottom = AnimRect.top+120;

AnimFrame++;
AnimFrame = AnimFrame & 15;

//Blt the offscreen surface to the back buffer
do
{
    ddreturn = lpBackbuffer->BltFast( XPos, YPos,
                                    lpOffscreen, &AnimRect,
                                    DDBLTFAST_SRCCOLORKEY|DDBLTFAST_WAIT );
    if ( ddreturn == DDERR_SURFACELOST )
    {
        if ( !RestoreSurfaces() )
            return FALSE;
    }
}

```

```

}while ( ddreturn != DD_OK );

//Save old positions for restoring on the next frame
XOld[Frame] = XPos;
YOld[Frame] = YPos;

//set of appropriate buffers for the next frame
if ( Frame == EVEN )
    Frame = ODD;
else
    Frame = EVEN;

//Let DirectDraw switch our surface pointers
do
{
    ddreturn = lpPrimary->Flip ( NULL, DDFLIP_WAIT );
    if ( ddreturn == DDERR_SURFACELOST )
    {
        if ( !RestoreSurfaces() )
            return FALSE;
    }
}while ( ddreturn != DD_OK );

return TRUE;

} //DrawFrame

/*****
//RestoreSurfaces - restores all lost surfaces
//
//returns:
//  BOOL True - success
*****/
BOOL RestoreSurfaces ( void )
{
    HRESULT ddreturn;

    ddreturn = lpPrimary->Restore();
    if ( ddreturn == DD_OK )
    {
        if ( !BMPToDirectDrawSurface (lpPrimary, stars.bmp" ) )
            return FALSE;
    }
    else
    {
        return FALSE;
    }
    ddreturn = lpBackbuffer->Restore();
    if ( ddreturn == DD_OK )
    {
        if ( !BMPToDirectDrawSurface (lpBackbuffer, "stars.bmp" ) )

```

```

        return FALSE;
    }else
    {
        return FALSE;
    }

    ddreturn = lpOffscreen->Restore();
    if( ddreturn == DD_OK )
    {
        if ( !BMPToDirectDrawSurface (lpOffscreen, "planet.bmp" ) )
            return FALSE;
    }else
    {
        return FALSE;
    }

    ddreturn = lpOffBufferEven->Restore();
    if( ddreturn != DD_OK )
        return FALSE;

    ddreturn = lpOffBufferOdd->Restore();
    if( ddreturn != DD_OK )
        return FALSE;

    return TRUE;
}

//RestoreSurfaces

//*****
//BMPToDirectDrawSurface - opens and copies a BMP to a DD surface
//input:
//  LPDIRECTDRAWSURFACE surface
//  char *bmp - filename
//returns:
//  BOOL TRUE - success
//*****
BOOL BMPToDirectDrawSurface( LPDIRECTDRAWSURFACE surface, char *bmp )
{
    HRESULT          ddreturn;
    DWORD            actualRead;
    HANDLE           hfile;
    char             path[MAX_PATH];
    BITMAPFILEHEADER BMPFileHead;
    BITMAPINFOHEADER BMPFileInfo;
    RGBQUAD          Palette[256];
    PALETTEENTRY      pe[256];
    LPDIRECTDRAWPALETTE lpDDPalette;
    DDSURFACEDESC     ddsd;
    int              i,src_width;
    BYTE             *lpDDMemory,*lpBMPMemory,*image;

```

```

hfile = CreateFile( bmp, GENERIC_READ, FILE_SHARE_READ,
                    (LPSECURITY_ATTRIBUTES)NULL,
                    OPEN_EXISTING, FILE_ATTRIBUTE_NORMAL,
                    (HANDLE) NULL );

if ( hfile == INVALID_HANDLE_VALUE )
{
    strcpy( path, "..\\");           //back up one directory
    strcat( path, bmp );
    hfile = CreateFile( path, GENERIC_READ, FILE_SHARE_READ,
                        (LPSECURITY_ATTRIBUTES)NULL,
                        OPEN_EXISTING, FILE_ATTRIBUTE_NORMAL,
                        (HANDLE) NULL );
    if ( hfile == INVALID_HANDLE_VALUE )
        return FALSE;
}

// Read the BMP header and info structures
if( !ReadFile ( hfile, &BMPFileHead, sizeof ( BMPFileHead ),
                &actualRead, NULL) )
    return FALSE;

if( !ReadFile ( hfile, &BMPFileInfo, sizeof ( BMPFileInfo ),
                &actualRead, NULL) )
    return FALSE;

//Make sure the BMP is 8-bit color
if ( BMPFileInfo.biBitCount != 8 )
    return FALSE;

//get the BMP palette
if ( !ReadFile ( hfile, Palette, sizeof( Palette ), &actualRead, NULL )
)
    return FALSE;

//Allocate memory for image data
image = (BYTE *) LocalAlloc ( LPTR,
                              BMPFileInfo.biWidth * BMPFileInfo.biHeight );
if( image == NULL )
    return FALSE;

//read BMP into memory
if( !ReadFile ( hfile, image,
                BMPFileInfo.biWidth * BMPFileInfo.biHeight,
                &actualRead, NULL) )
{
    LocalFree( image );
    return FALSE;
}

```

```

//copy BMP data to DirectDraw surface
memset ( &ddsd, 0, sizeof( DDSURFACEDESC ) );
ddsd.dwSize = sizeof( ddsd );
ddreturn = surface->Lock( NULL, &ddsd, 0, NULL );

if( ddreturn != DD_OK )
{
    LocalFree( image );
    return FALSE;
}

lpDDMemory = (BYTE *) ddsd.lpSurface;
lpBMPMemory = image+((BMPFileInfo.biHeight-1)*BMPFileInfo.biWidth);

if ( BMPFileInfo.biWidth > ddsd.lPitch )
    src_width = ddsd.lPitch;
else
    src_width = BMPFileInfo.biWidth;

for( i = 0; i < BMPFileInfo.biHeight; i++ )
{
    memcpy ( lpDDMemory, lpBMPMemory, src_width );
    lpDDMemory += ddsd.lPitch;
    lpBMPMemory -= src_width;
}

surface->Unlock ( NULL );
LocalFree( image );
CloseHandle(hfile);

//create a DirectDraw palette for the surface
for( i=0; i<256; i++ )
{
    pe[i].peRed = Palette[i].rgbRed;
    pe[i].peGreen = Palette[i].rgbGreen;
    pe[i].peBlue = Palette[i].rgbBlue;
}
ddreturn = lpDirectDrawObject->CreatePalette( DDPCAPS_8BIT,
                                                pe,
                                                &lpDDPalette,
                                                NULL );

if ( ddreturn != DD_OK )
    return FALSE;
surface->SetPalette ( lpDDPalette );

return TRUE;
} //BMPToDirectDrawSurface

```

In **WinMain**, we establish our mandatory links to DirectDraw, set the desired video mode, and create our surfaces. We then apply our source color key, **ddck**, to

the off-screen surface. The **BMPToDirectDrawSurface** function is then used to initialize the background surfaces with the STARS.BMP image and the off-screen surface is filled with the PLANET.BMP data.

If you load the PLANET.BMP file into a paint program, you can't avoid noticing the rather jarring pink color that is used to signify the transparent pixel value. Since this palette color index is never seen by the user, it could be set to any color. Using a color like this simply helps the software designer clearly delineate transparent and nontransparent pixels.

The DrawFrame function implements a simple, timed animation of the planet graphic. In this example, the individual frames that create the rotating planet can be stored as one large off-screen surface. Although this method of storing animation frames is easy to implement, it isn't always efficient in terms of memory usage.

THE TB_DEST.CPP PROGRAM

The following program modifies TB_SRC.CPP to demonstrate a transparent blt using a destination color key. It should be noted that some video cards can't support destination color keys (source color keying is supported by almost all DirectDraw drivers). If your video card can't support destination color keys, this program won't run.

```
//*****
//DirectDraw Programming
//Bret M. Timmins
//
//(C)1995 M&T Books
//
//tb_dest.cpp
// Demonstrates transparent blting using blt fast
// with a destination color key
//*****

#define WIN32_LEAN_AND_MEAN
#include <windows.h>
#include <windowsx.h>

#ifdef __BORLANDC__ && defined( __WIN32__ )
#define _WIN32
#endif

#include "..\gdk\sdk\inc\ddraw.h"
```

DIRECTDRAW PROGRAMMING

Chapter Eight: Blting: Transparent Blting

```
#include <stdio.h>
#include <stdlib.h>
#include <stdarg.h>

#define EVEN 0
#define ODD 1
#define NEG 0xffffffff

LPDIRECTDRAW          lpDirectDrawObject;    // DirectDraw object
LPDIRECTDRAWSURFACE   lpPrimary;             // DirectDraw primary
surface
LPDIRECTDRAWSURFACE   lpBackbuffer;          // DirectDraw primary
surface
LPDIRECTDRAWSURFACE   lpOffscreen;           // Offscreen surface
LPDIRECTDRAWSURFACE   lpOffBufferEven;       // Offscreen surface,
buffers                                           // pixel data that would be
                                                // destroyed by BltFast for
                                                // the primary surface

LPDIRECTDRAWSURFACE   lpOffBufferOdd;         // Offscreen surface,
buffers                                           // pixel data that would be
                                                // destroyed by BltFast for
                                                // the back buffer surface

BOOL                  ActiveApp;              //Is this program active?

int PASCAL WinMain( HINSTANCE hInstance, HINSTANCE hPrevInstance,
                    LPSTR lpCmdLine, int nCmdShow);
long FAR PASCAL WindowProc ( HWND hwnd, UINT message,
                             WPARAM wParam, LPARAM lParam );

BOOL DrawFrame ( void );
BOOL RestoreSurfaces ( void );
BOOL BMPToDirectDrawSurface( LPDIRECTDRAWSURFACE surface, char *bmp );

//*****
//WinMain - mandatory windows init function
//
//*****
int PASCAL WinMain( HINSTANCE hInstance, HINSTANCE hPrevInstance,
                    LPSTR lpCmdLine, int nCmdShow)
{
    MSG          msg;
    HWND         hwnd;
    WNDCLASS     wc;
    static char ClassName[] = "DEST_BLT";
    DDSURFACEDESC ddsd;
    DDSCAPS      ddscaps;
    DDCOLORKEY   ddck;
    HRESULT      ddreturn;

```



```
lpCmdLine = lpCmdLine;
hPrevInstance = hPrevInstance;

//register and realize our display window
wc.style = CS_HREDRAW | CS_VREDRAW;
wc.lpfWndProc = WindowProc;
wc.cbClsExtra = 0;
wc.cbWndExtra = 0;
wc.hInstance = hInstance;
wc.hIcon = LoadIcon( hInstance, IDI_APPLICATION );
wc.hCursor = LoadCursor( NULL, IDC_ARROW );
wc.hbrBackground = NULL;
wc.lpszMenuName = ClassName;
wc.lpszClassName = ClassName;
RegisterClass( &wc );

//create a full screen window so that GDI won't ever be called
hwnd = CreateWindowEx(
    WS_EX_TOPMOST,
    ClassName,
    ClassName,
    WS_POPUP,
    0,
    0,
    GetSystemMetrics(SM_CXSCREEN),
    GetSystemMetrics(SM_CYSCREEN),
    NULL,
    NULL,
    hInstance,
    NULL );

if ( !hwnd )
    return FALSE;

ShowWindow( hwnd, nCmdShow );
UpdateWindow( hwnd );
SetFocus( hwnd );
ShowCursor( FALSE );

//Instantiate our DirectDraw object
ddreturn = DirectDrawCreate( NULL,
                             &lpDirectDrawObject, NULL );
if ( ddreturn != DD_OK )
{
    DestroyWindow( hwnd );
    return FALSE;
}

ddreturn = lpDirectDrawObject->SetCooperativeLevel( hwnd,
    DDSCL_EXCLUSIVE | DDSCL_FULLSCREEN );
```

```
if ( ddreturn != DD_OK )
{
    DestroyWindow( hwnd );
    return FALSE;
}

// Set the video mode to 640x480x8
ddreturn = lpDirectDrawObject->SetDisplayMode( 640, 480, 8);
if ( ddreturn != DD_OK )
{
    DestroyWindow( hwnd );
    return FALSE;
}

// Create the primary surface and one back buffer surface
ddsd.dwSize = sizeof( ddsd );
ddsd.dwFlags = DDSD_CAPS | DDSD_BACKBUFFERCOUNT;
ddsd.ddsCaps.dwCaps = DDSCAPS_PRIMARYSURFACE |
                    DDSCAPS_FLIP |
                    DDSCAPS_COMPLEX;
ddsd.dwBackBufferCount = 1;
ddreturn = lpDirectDrawObject->CreateSurface( &ddsd, &lpPrimary, NULL
);
    if ( ddreturn != DD_OK )
    {
        DestroyWindow( hwnd );
        return FALSE;
    }

//get a surface pointer to our back buffer
ddscaps.dwCaps = DDSCAPS_BACKBUFFER;
ddreturn = lpPrimary->GetAttachedSurface ( &ddscaps, &lpBackbuffer);
if( ddreturn != DD_OK )
{
    DestroyWindow ( hwnd );
    return FALSE;
}

// Create the main offscreen surface
ddsd.dwFlags = DDSD_CAPS | DDSD_HEIGHT | DDSD_WIDTH;
ddsd.ddsCaps.dwCaps = DDSCAPS_OFFSCREENPLAIN;
ddsd.dwHeight = 120;
ddsd.dwWidth = 128;
ddreturn = lpDirectDrawObject->CreateSurface ( &ddsd,
                                                &lpOffscreen, NULL );

    if ( ddreturn != DD_OK )
    {
        DestroyWindow( hwnd );
        return FALSE;
    }
}
```

```
// Create the offscreen surface buffer EVEN
ddsd.dwFlags = DDSD_CAPS | DDSD_HEIGHT | DDSD_WIDTH;
ddsd.ddsCaps.dwCaps = DDSCAPS_OFFSCREENPLAIN;
ddsd.dwHeight = 120;
ddsd.dwWidth = 128;
ddreturn = lpDirectDrawObject->CreateSurface ( &ddsd,
                                                &lpOffBufferEven, NULL );
if ( ddreturn != DD_OK )
{
    DestroyWindow( hwnd );
    return FALSE;
}

// Create the offscreen surface buffer ODD
ddsd.dwFlags = DDSD_CAPS | DDSD_HEIGHT | DDSD_WIDTH;
ddsd.ddsCaps.dwCaps = DDSCAPS_OFFSCREENPLAIN;
ddsd.dwHeight = 120;
ddsd.dwWidth = 128;
ddreturn = lpDirectDrawObject->CreateSurface ( &ddsd,
                                                &lpOffBufferOdd, NULL );
if ( ddreturn != DD_OK )
{
    DestroyWindow ( hwnd );
    return FALSE;
}

//Set a destination color key for the back buffer
ddck.dwColorSpaceLowValue = 0;
ddck.dwColorSpaceHighValue = 0;
lpBackbuffer->SetColorKey( DDCKEY_DESTBLT, &ddck );

//copy a BMP file to the primary surface
if ( !BMPToDirectDrawSurface (lpPrimary, "marble.bmp" ) )
{
    DestroyWindow ( hwnd );
    return FALSE;
}

//copy to the back buffer
if ( !BMPToDirectDrawSurface (lpBackbuffer, "marble.bmp" ) )
{
    DestroyWindow ( hwnd );
    return FALSE;
}

//copy to the offscreen surface
if ( !BMPToDirectDrawSurface (lpOffscreen, "tile.bmp" ) )
{
    DestroyWindow ( hwnd );
    return FALSE;
}
```

```
    }

while( *1 )
{
    if( PeekMessage( &msg, NULL, 0, 0, PM_NOREMOVE ) )
    {
        if( !GetMessage( &msg, NULL, 0, 0 ) )
            return msg.wParam;
        TranslateMessage(&msg);
        DispatchMessage(&msg);
    }else if ( ActiveApp )
    {
        DrawFrame ();
    }else
    {
        WaitMessage ();
    }
}

} //WinMain

//*****
//WindowProc - receive and handle windows messages
//
//
//*****
long FAR PASCAL WindowProc( HWND hwnd, UINT message,
                           WPARAM wParam, LPARAM lParam )
{
    switch( message )
    {
        case WM_ACTIVATEAPP:
            ActiveApp = wParam;
            break;

        case WM_CREATE:
            break;

        case WM_KEYDOWN:
            switch ( wParam )
            {
                case VK_ESCAPE:
                    DestroyWindow ( hwnd );
                    break;
            }
            break;

        case WM_DESTROY:
            if ( lpDirectDrawObject != NULL )
            {

```

```

        if ( lpOffBufferEven != NULL )
            lpOffBufferEven->Release();
        if ( lpOffBufferOdd != NULL )
            lpOffBufferOdd->Release();
        if ( lpOffscreen != NULL )
            lpOffscreen->Release();
        if ( lpPrimary != NULL )
            lpPrimary->Release();
        lpDirectDrawObject->Release();
    }

    ShowCursor ( TRUE );
    PostQuitMessage ( wParam );
    break;

default:
    return DefWindowProc ( hwnd, message, wParam, lParam );
}

return 0L;

} //WindowProc

//*****
//DrawFrame - Compose a frame of graphics, flip surfaces
//
//return:
//    BOOL TRUE - success
//*****
BOOL DrawFrame ( void )
{
    HRESULT ddreturn;
    static RECT rect = { 0, 0, 128, 120 };
    RECT BufferRect;
    static DWORD XPos = 0, YPos = 0;
    static DWORD XOld[2] = { NEG, NEG };
    static DWORD YOld[2] = { NEG, NEG };
    static int XDelta = 2, YDelta = 2;
    static BOOL Frame = EVEN;
    static LPDIRECTDRAWSURFACE lpBuffer;
    static DWORD TickCount = GetTickCount();

    //simple timing
    if ( (GetTickCount() - TickCount) < 25 )
        return TRUE;
    TickCount = GetTickCount();

    if ( Frame == EVEN )
        lpBuffer = lpOffBufferEven;
    else

```

```
    lpBuffer = lpOffBufferOdd;

//Restore back buffer pixels from previous frame
if ( X01d[Frame] != NEG )
{
    do
    {
        ddreturn = lpBackbuffer->BltFast( X01d[Frame], Y01d[Frame],
            lpBuffer, &rect,
            DDBLTFAST_NOCOLORKEY|DDBLTFAST_WAIT );
        if ( ddreturn == DDERR_SURFACELOST )
        {
            if ( !RestoreSurfaces() )
                return FALSE;
            //case-after restore, buffers are invalid, rebuild them
            X01d[0] = NEG; X01d[1] = NEG;
            Y01d[0] = NEG; Y01d[1] = NEG;
            break;
        }
    }while ( ddreturn != DD_OK );
}

//Save back buffer pixels to the offscreen buffer
BufferRect.left = XPos;
BufferRect.top = YPos;
BufferRect.right = XPos+128;
BufferRect.bottom = YPos+120;
do
{
    ddreturn = lpBuffer->BltFast( 0, 0,
        lpBackbuffer, &BufferRect,
        DDBLTFAST_NOCOLORKEY|DDBLTFAST_WAIT );
    if ( ddreturn == DDERR_SURFACELOST )
    {
        if ( !RestoreSurfaces() )
            return FALSE;
    }
}while ( ddreturn != DD_OK );

//Blt the offscreen surface to the back buffer
do
{
    ddreturn = lpBackbuffer->BltFast( XPos, YPos,
        lpOffscreen, &rect,
        DDBLTFAST_DESTCOLORKEY|DDBLTFAST_WAIT );
    if ( ddreturn == DDERR_SURFACELOST )
    {
        if ( !RestoreSurfaces() )
            return FALSE;
    }
}
```

```
}while ( ddreturn != DD_OK );

//Save old positions for restoring on the next frame
XOld[Frame] = XPos;
YOld[Frame] = YPos;

//set of appropriate buffers for the next frame
if ( Frame == EVEN )
    Frame = ODD;
else
    Frame = EVEN;

//Adjust destination coordinates for next frame
if ( (XPos+XDelta) > 640-128 )
{
    XDelta = -XDelta;
    XPos += XDelta;
}
if ( (YPos+YDelta) > 480-120 )
{
    YDelta = -YDelta;
    YPos += YDelta;
}
//Let DirectDraw switch our surface pointers
do
{
    ddreturn = lpPrimary->Flip ( NULL, DDFLIP_WAIT );
    if ( ddreturn == DDERR_SURFACELOST )
    {
        if ( !RestoreSurfaces() )
            return FALSE;
    }
}while ( ddreturn != DD_OK );

return TRUE;

} //DrawFrame

//*****
//RestoreSurfaces - restores all lost surfaces
//
//returns:
//    BOOL True - success
//*****
BOOL RestoreSurfaces ( void )
{
    HRESULT ddreturn;

    ddreturn = lpPrimary->Restore();
    if ( ddreturn == DD_OK )
```

```

{
    if ( !BMPToDirectDrawSurface (lpPrimary, "marble.bmp" ) )
        return FALSE;
}
else
{
    return FALSE;
}
ddreturn = lpBackbuffer->Restore();
if( ddreturn == DD_OK )
{
    if ( !BMPToDirectDrawSurface (lpBackbuffer, "marble.bmp" ) )
        return FALSE;
}
else
{
    return FALSE;
}

ddreturn = lpOffscreen->Restore();
if( ddreturn == DD_OK )
{
    if ( !BMPToDirectDrawSurface (lpOffscreen, "tile.bmp" ) )
        return FALSE;
}
else
{
    return FALSE;
}

ddreturn = lpOffBufferEven->Restore();
if( ddreturn != DD_OK )
    return FALSE;

ddreturn = lpOffBufferOdd->Restore();
if( ddreturn != DD_OK )
    return FALSE;

return TRUE;
}

//RestoreSurfaces

//*****
//BMPToDirectDrawSurface - opens and copies a BMP to a DD surface
//input:
//    LPDIRECTDRAW_SURFACE surface
//    char *bmp - filename
//returns:
//    BOOL TRUE - success
//*****
BOOL BMPToDirectDrawSurface( LPDIRECTDRAW_SURFACE surface, char *bmp )
{
    HRESULT                ddreturn;

```



```

DWORD          actualRead;
HANDLE          hfile;
char            path[MAX_PATH];
BITMAPFILEHEADER BMPFileHead;
BITMAPINFOHEADER BMPFileInfo;
RGBQUAD         Palette[256];
PALETTEENTRY    pe[256];
LPDIRECTDRAWPALETTE lpDDPalette;
DDSURFACEDESC   ddsd;
int             i,src_width;
BYTE            *lpDDMemory,*lpBMPMemory,*image;

hfile = CreateFile( bmp, GENERIC_READ, FILE_SHARE_READ,
                    (LPSECURITY_ATTRIBUTES)NULL,
                    OPEN_EXISTING, FILE_ATTRIBUTE_NORMAL,
                    (HANDLE) NULL );

if ( hfile == INVALID_HANDLE_VALUE )
{
    strcpy (path,"..\\");
    directory
    strcat( path, bmp );
    hfile = CreateFile( path, GENERIC_READ, FILE_SHARE_READ,
                        (LPSECURITY_ATTRIBUTES)NULL,
                        OPEN_EXISTING, FILE_ATTRIBUTE_NORMAL,
                        (HANDLE) NULL );
    if ( hfile == INVALID_HANDLE_VALUE )
        return FALSE;
}

// Read the BMP header and info structures
if( !ReadFile ( hfile, &BMPFileHead, sizeof ( BMPFileHead ),
                &actualRead, NULL) )
    return FALSE;

if( !ReadFile ( hfile, &BMPFileInfo, sizeof ( BMPFileInfo ),
                &actualRead, NULL) )
    return FALSE;

//Make sure the BMP is 8-bit color
if ( BMPFileInfo.biBitCount != 8 )
    return FALSE;

//get the BMP palette
if ( !ReadFile ( hfile, Palette, sizeof( Palette ), &actualRead, NULL )
)
    return FALSE;

//Allocate memory for image data
image = (BYTE *) LocalAlloc ( LPTR,

```

```
        BMPFileInfo.biWidth * BMPFileInfo.biHeight );

if( image == NULL )
    return FALSE;

    //read BMP into memory
    if( !ReadFile ( hfile, image,
        BMPFileInfo.biWidth * BMPFileInfo.biHeight,
        &actualRead, NULL) )
    {
        LocalFree( image );
        return FALSE;
    }

    //copy BMP data to DirectDraw surface
    memset ( &ddsd, 0, sizeof( DDSURFACEDESC ) );
    ddsd.dwSize = sizeof( ddsd );
    ddreturn = surface->Lock( NULL, &ddsd, 0, NULL );

    if( ddreturn != DD_OK )
    {
        LocalFree( image );
        return FALSE;
    }

    lpDDMemory = (BYTE *) ddsd.lpSurface;
    lpBMPMemory = image+((BMPFileInfo.biHeight-1)*BMPFileInfo.biWidth);

    if ( BMPFileInfo.biWidth > ddsd.lPitch )
        src_width = ddsd.lPitch;
    else
        src_width = BMPFileInfo.biWidth;

    for( i = 0; i < BMPFileInfo.biHeight; i++ )
    {
        memcpy ( lpDDMemory, lpBMPMemory, src_width );
        lpDDMemory += ddsd.lPitch;
        lpBMPMemory -= src_width;
    }

    surface->Unlock ( NULL );
    LocalFree( image );
    CloseHandle(hfile);

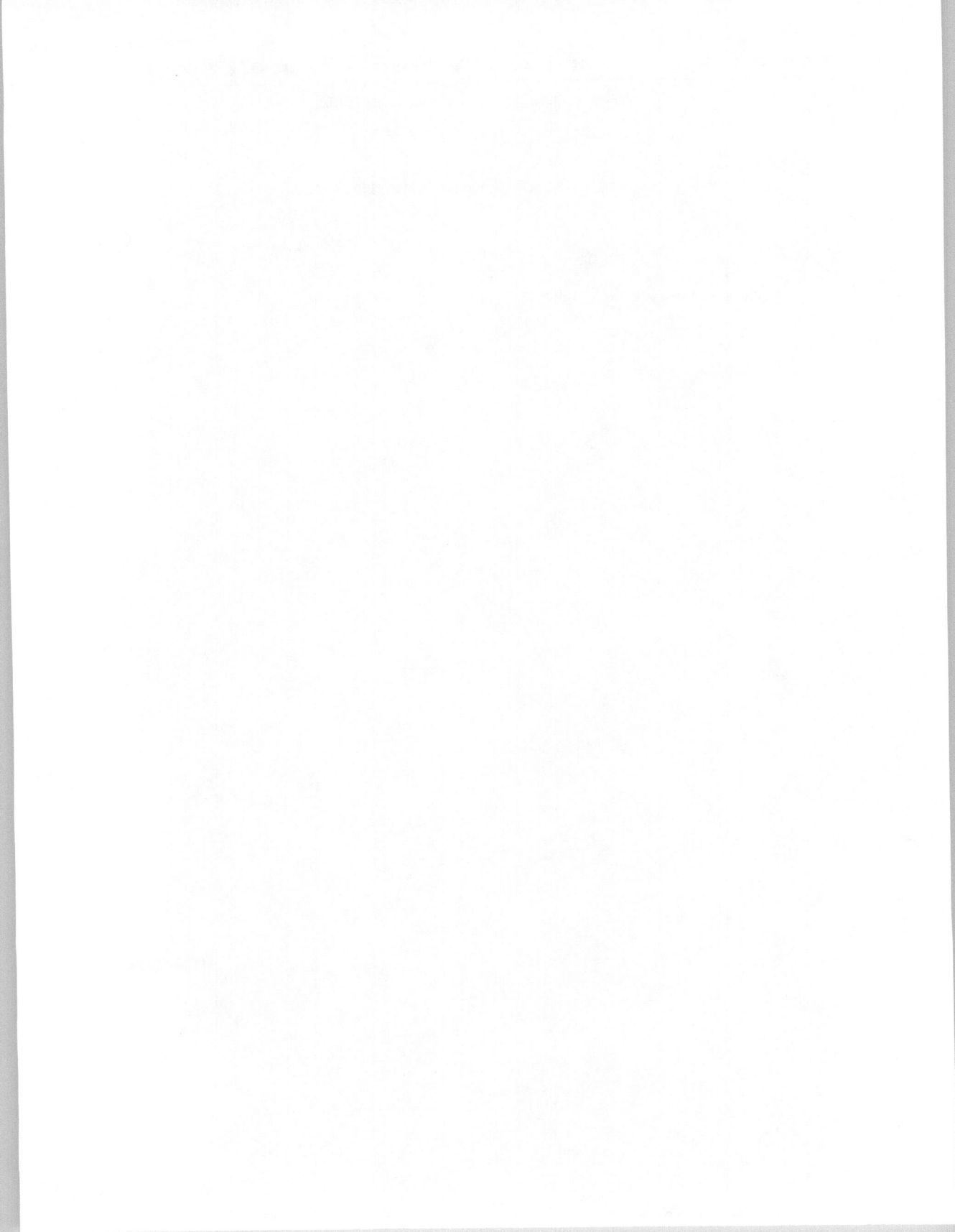
    //create a DirectDraw palette for the surface
    for( i=0; i<256; i++ )
    {
        pe[i].peRed = Palette[i].rgbRed;
        pe[i].peGreen = Palette[i].rgbGreen;
        pe[i].peBlue = Palette[i].rgbBlue;
    }
```

```
ddreturn = lpDirectDrawObject->CreatePalette( DDPCAPS_8BIT,
                                                pe,
                                                &lpDDPalette,
                                                NULL );

if ( ddreturn != DD_OK )
    return FALSE;
surface->SetPalette ( lpDDPalette );

return TRUE;

} //BMPToDirectDrawSurface
```



Blting: Blt Effects

In the last chapter, we covered the most fundamental of blt special effects, the transparent blt. In this chapter we will introduce the versatile **Blt** member function and see how it can be used to produce a large number of graphic effects.

BLT OR BLTFAST?

As we have seen in the last three chapters, the **BltFast** function is used to perform basic blting of pixel data from one surface to another. The **BltFast** function can perform a straight pixel-to-pixel copy, or when combined with a source or a destination color key, it can produce a transparent blt.

BltFast is designed to set up the blter hardware on the video card for extremely quick pixel move operations. If a blter chip is not present, **BltFast** will execute the blt using optimized software routines. **BltFast** is a narrowly focused function, designed to perform a limited set of operations as fast as possible. For more advanced types of blting, **DirectDraw** provides the **Blt** function.

The **Blt** routine is a general-purpose function, capable of shrinking and stretch blting, rotating pixel data, color fills, image clipping, and transparent blting using source or destination color keys, among others things. Like the **BltFast** function, **Blt** will use the hardware blter chip if it can support the requested blt operation. If the hardware support is lacking, **Blt** can emulate most operations through software.

In deciding whether to use **Blt** or **BltFast** for a given blt operation, it is helpful to keep in mind their relative strengths and the overriding goal of all

game programmers, speed. If the blt is a straight pixel-to-pixel copy or a simple transparent pixel copy, use **BltFast**. More complex types of blting will require the **Blt** function.

THE BLT FUNCTION

The syntax of the **Blt** function is more complicated than **BltFast**. Several blt options require the use of an input structure, **DDBLTFX**, which will be discussed shortly.

```
HRESULT Blt(
    LPDIRECTDRAW SURFACE lpDDSurface,
    LPRECT lpDestRect,
    LPDIRECTDRAW SURFACE lpDDSrcSurface,
    LPRECT lpSrcRect,
    DWORD dwFlags,
    LPDDBLTFX lpDDBltFx)
```

lpDDSurface points to the surface that will be the destination of the blt operation. **lpDestRect** points to a Windows RECT structure that defines a portion of the destination surface that will receive the blted pixels. This syntax is different from the **BltFast** function where the destination coordinates are defined by the **dwX** and **dwY** parameters and the width and height of the source RECT structure. **lpDDSrcSurface** points to the source drawing surface for the blt. **lpSrcRect** points to a RECT structure that determines which portion of the source surface will be blted. The **Blt** function compares the two RECT structures, and if they are different, **Blt** will perform a shrinking or stretching of the pixels, as appropriate. The method of shrinking and stretching is very similar to the **StretchBlt** and **StretchDIBBlt** functions in the Windows API. The **dwFlags** parameter tells **Blt** the type of operation that is to be performed:

```
DDBLT_COLORFILL
```

This tells **Blt** to fill the destination surface with the color that is defined in the **dwFillColor** field of the **DDBLTFX** structure. See the color fill example within this chapter.

```
DDBLT_FX
```

This flag informs **Blt** that the **dwDDFX** field of the **DDBLTFX** structure is valid. If this flag is encountered, **Blt** will look entirely to the **DDBLTFX** structure for

information on how to perform the requested blt. See the **DDBLT_FX** heading later in this chapter.

DDBLT_ROPS

This flag tells **Blt** that the **dwDDROPS** field of **DDBLTFX** contains a custom ROP (raster operation) that should be used in the blt operation. A Windows raster operation is a code that determines how source and destination pixels will be combined during the blt, usually using Boolean operators in some form. Consult the Win32 API documentation for help on ROPs.

DDBLT_KEYDEST

This flag tells **Blt** to use the attached destination color key to produce a transparent blt.

DDBLT_KEYDESTOVERRIDE

This flag tells **Blt** to use the value in the **dckDestColorKey** field of the **DDBLTFX** structure as the destination color key for this blt. As its name implies, **DDBLT_KEYDESTOVERRIDE** will supersede any attached destination color key.

DDBLT_KEYSRC

This flag tells **Blt** to use the attached source color key to produce a transparent blt.

DDBLT_SCROVERRIDE

This flag uses the value in the **dckDestColorKey** field of the **DDBLTFX** structure as the source color key for this blt operation.

DDBLT_ROP

This flag tells **Blt** that the **dwRop** field in the **DDBLTFX** structure contains a predefined ROP code that should be used in the blt operation. A list of common Windows ROPs can be found in the Win32 API help files coverage of the BitBlt function.

DDBLT_ROTATIONANGLE

If this flag is set, **Blt** will use the **dwRotationAngle** field in the **DDBLTFX** structure to rotate the pixels during the blt operation in increments of 1/100th of a degree.

DDBLT_WAIT

This flag tells **Blt** to wait until the blter chip is free to perform the blt operation, and not to return on a **DDERR_WASSTILLDRAWING** condition. The usage of **DDBLT_WAIT** is identical to **BltFast**'s **DDBLTFAST_WAIT** flag. **Blt** uses additional flags for alpha and Z buffer surfaces. A full reference can be found in Appendix A.

The last parameter of **Blt**, **lpDDBltFx**, points to a **DDBLTFX** structure. If this structure is not required for the blt operation, such as blt using an attached source color key, **lpDDBltFx** should be set to **NULL**.

DDBLT_FX

As we have seen earlier, the **DDBLTFX** structure is used to pass information to the **Blt** function concerning how a given blt operation should be performed. Additional blt effects are available through the **dwDDFX** field of the **DDBLTFX** structure:

DDBLT_MIRRORLEFTRIGHT

If this value is set in **dwDDFX**, **Blt** will mirror the pixels in the source surface from left to right when blted to the destination surface.

DDBLT_MIRRORUPDOWN

This performs the blt mirroring the source pixels along the *x* axis.

DDBLT_ROTATE180

This flips the source pixels upside down.

DDBLT_ROTATE270

This rotates the source pixels 270°.

DDBLT_ROTATE90

This rotates the source pixels 90°.

A full reference to the **DDBLTFX** structure can be found in Appendix A.

THE COLOR FILL

Most blter chips are capable of performing an extremely fast color fill. A color fill is the act of writing a solid color to a drawing surface. Fast color fill operations are especially useful in 3D applications where surfaces must be “cleared” with a background color before the next frame of graphics can be drawn. The following source code fragment sets up a color fill using the **Blt** function:

```
LPDIRECTDRAWSURFACE lpBackbuffer;
DDBLTFX BltFX;
BltFX.dwSize = sizeof(BltFX);
BltFX.dwFillColor = 1;
lpBackBuffer->Blt ( NULL,
    NULL,
    NULL,
    DDBLT_COLORFILL,
    &BltFX);
```

A NULL value in the **lpDestRect** parameter tells **Blt** to write to the entire drawing surface. Specific regions of the destination surface can be filled by passing a legal RECT in **lpDestRect**. Since the color fill operation doesn’t involve a source drawing surface, the **lpSrcSurface** pointer and the **lpSrcRect** pointer are also set to NULL. The color value in **BltFX.dwFillColor** will resolve to a palette index for 8-bit video modes or an actual color value for 16-bit color modes.

THE SHRINK.CPP PROGRAM

The following program uses the **Blt** function to dynamically shrink an off-screen surface image.

```
/******
//DirectDraw Programming
//Bret Timmins
//
//(C)1995 M&T Books
//
//SHRINK.CPP
// Demonstrates basic usage of the Blt function
```

```
//
//*****

#define WIN32_LEAN_AND_MEAN
#include <windows.h>
#include <windowsx.h>

#if defined( __BORLANDC__ ) && defined( __WIN32__ )
#define _WIN32
#endif

#include "gdk\sdk\inc\ddraw.h"
#include <stdio.h>
#include <stdlib.h>
#include <stdarg.h>

LPDIRECTDRAW          lpDirectDrawObject;      // DirectDraw object
LPDIRECTDRAWSURFACE   lpPrimary;              // DirectDraw primary
surface
LPDIRECTDRAWSURFACE   lpBackbuffer;           // DirectDraw primary
surface
LPDIRECTDRAWSURFACE   lpOffscreen;            // Offscreen surface
LPDIRECTDRAWSURFACE   lpOffBuffer;            // Offscreen surface,
buffers
// pixel data that would be
// destroyed by Blt

BOOL                  ActiveApp;              // Is this program active?

int PASCAL WinMain( HINSTANCE hInstance, HINSTANCE hPrevInstance,
                    LPSTR lpCmdLine, int nCmdShow);
long FAR PASCAL WindowProc ( HWND hwnd, UINT message,
                             WPARAM wParam, LPARAM lParam );

BOOL DrawFrame ( void );
BOOL RestoreSurfaces ( void );
BOOL BMPToDirectDrawSurface( LPDIRECTDRAWSURFACE surface, char *bmp );

//*****
//WinMain - mandatory windows init function
//
//*****
int PASCAL WinMain( HINSTANCE hInstance, HINSTANCE hPrevInstance,
                    LPSTR lpCmdLine, int nCmdShow)
{
    MSG          msg;
    HWND         hwnd;
    WNDCLASS     wc;
    static char  ClassName[] = "Shrink";
    DDSURFACEDESC ddsd;
    DDSCAPS      ddscaps;
    DDCOLORKEY   ddck;
```

```
HRESULT      ddreturn;

lpCmdLine = lpCmdLine;
hPrevInstance = hPrevInstance;

//register and realize our display window
wc.style = CS_HREDRAW | CS_VREDRAW;
wc.lpfnWndProc = WindowProc;
wc.cbClsExtra = 0;
wc.cbWndExtra = 0;
wc.hInstance = hInstance;
wc.hIcon = LoadIcon( hInstance, IDI_APPLICATION );
wc.hCursor = LoadCursor( NULL, IDC_ARROW );
wc.hbrBackground = NULL;
wc.lpszMenuName = ClassName;
wc.lpszClassName = ClassName;
RegisterClass( &wc );

//create a full screen window so that GDI won't ever be called
hwnd = CreateWindowEx(
    WS_EX_TOPMOST,
    ClassName,
    ClassName,
    WS_POPUP,
    0,
    0,
    GetSystemMetrics(SM_CXSCREEN),
    GetSystemMetrics(SM_CYSCREEN),
    NULL,
    NULL,
    hInstance,
    NULL );

if( !hwnd )
    return FALSE;

ShowWindow( hwnd, nCmdShow );
UpdateWindow( hwnd );
SetFocus( hwnd );
ShowCursor( FALSE );

//Instanciate our DirectDraw object
ddreturn = DirectDrawCreate( NULL,
                             &lpDirectDrawObject, NULL );

if ( ddreturn != DD_OK )
{
    DestroyWindow( hwnd );
    return FALSE;
}
```

```
ddreturn = lpDirectDrawObject->SetCooperativeLevel( hwnd,
                                                    DDSCL_EXCLUSIVE | DDSCL_FULLSCREEN );
if ( ddreturn != DD_OK )
{
    DestroyWindow( hwnd );
    return FALSE;
}

// Set the video mode to 640x480x8
ddreturn = lpDirectDrawObject->SetDisplayMode( 640, 480, 8);
if ( ddreturn != DD_OK )
{
    DestroyWindow( hwnd );
    return FALSE;
}

// Create the primary surface and one back buffer surface
ddsd.dwSize = sizeof( ddsd );
ddsd.dwFlags = DDSD_CAPS | DDSD_BACKBUFFERCOUNT;
ddsd.ddsCaps.dwCaps = DDSCAPS_PRIMARYSURFACE |
                    DDSCAPS_FLIP |
                    DDSCAPS_COMPLEX;
ddsd.dwBackBufferCount = 1;
ddreturn = lpDirectDrawObject->CreateSurface( &ddsd, &lpPrimary, NULL
);
if ( ddreturn != DD_OK )
{
    DestroyWindow( hwnd );
    return FALSE;
}

//get a surface pointer to our back buffer
ddscaps.dwCaps = DDSCAPS_BACKBUFFER;
ddreturn = lpPrimary->GetAttachedSurface ( &ddscaps, &lpBackbuffer);
if( ddreturn != DD_OK )
{
    DestroyWindow ( hwnd );
    return FALSE;
}

// Create the main offscreen surface
ddsd.dwFlags = DDSD_CAPS | DDSD_HEIGHT | DDSD_WIDTH;
ddsd.ddsCaps.dwCaps = DDSCAPS_OFFSCREENPLAIN;
ddsd.dwHeight = 190;
ddsd.dwWidth = 200;
ddreturn = lpDirectDrawObject->CreateSurface ( &ddsd,
                                                &lpOffscreen, NULL );
if ( ddreturn != DD_OK )
{
    DestroyWindow( hwnd );
}
```

```

        return FALSE;
    }

    // Create the offscreen surface buffer
    ddsd.dwFlags = DDSD_CAPS | DDSD_HEIGHT | DDSD_WIDTH;
    ddsd.ddsCaps.dwCaps = DDSCAPS_OFFSCREENPLAIN;
    ddsd.dwHeight = 190;
    ddsd.dwWidth = 200;
    ddreturn = lpDirectDrawObject->CreateSurface ( &ddsd,
                                                    &lpOffBuffer, NULL );

    if ( ddreturn != DD_OK )
    {
        DestroyWindow( hwnd );
        return FALSE;
    }

    //copy a BMP file to the primary surface
    if ( !BMPToDirectDrawSurface (lpPrimary, "back.bmp" ) )
    {
        DestroyWindow ( hwnd );
        return FALSE;
    }

    //copy to the back buffer
    if ( !BMPToDirectDrawSurface (lpBackbuffer, "back.bmp" ) )
    {
        DestroyWindow ( hwnd );
        return FALSE;
    }

    //Set the transparent color key for the off screen surface
    ddck.dwColorSpaceLowValue = 0;
    ddck.dwColorSpaceHighValue = 0;
    lpOffscreen->SetColorKey( DDCKEY_SRCBLT, &ddck );

    //copy to the offscreen surface
    if ( !BMPToDirectDrawSurface (lpOffscreen, "star.bmp" ) )
    {
        DestroyWindow ( hwnd );
        return FALSE;
    }

    //re-fill the off screen buffer with data from the back buffer
    //so we don't have to worry about this in DrawFrame
    RECT BufferRect;

    BufferRect.left = (640/2)-(200/2);
    BufferRect.top = (480/2)-(190/2);
    BufferRect.right = BufferRect.left+200;

```

```

    BufferRect.bottom = BufferRect.top+190;
    do
    {
        ddreturn = lpOffBuffer->BltFast( 0, 0,
                                         lfe ());
    }
}

}

//WinMain

//*****
//WindowProc - receive and handle windows messages
//
//*****
long FAR PASCAL WindowProc( HWND hwnd, UINT message,
                           WPARAM wParam, LPARAM lParam )
{
    switch( message )
    {
        case WM_ACTIVATEAPP:
            ActiveApp = wParam;
            break;

        case WM_CREATE:
            break;

        case WM_KEYDOWN:
            switch ( wParam )
            {
                case VK_ESCAPE:
                    DestroyWindow ( hwnd );
                    break;
            }
            break;

        case WM_DESTROY:
            if ( lpDirectDrawObject != NULL )
            {
                if ( lpOffBuffer != NULL )
                    lpOffBuffer->Release();
                if ( lpOffscreen != NULL )
                    lpOffscreen->Release();
                if ( lpPrimary != NULL )
                    lpPrimary->Release();
                lpDirectDrawObject->Release();
            }

            ShowCursor ( TRUE );
            PostQuitMessage ( wParam );
            break;
    }
}

```

```

    default:
        return DefWindowProc ( hwnd, message, wParam, lParam );
    }

    return 0L;

} //WindowProc

//*****
//DrawFrame - Compose a frame of graphics, flip surfaces
//
//return:
//  BOOL TRUE - success
//*****
BOOL DrawFrame ( void )
{
    HRESULT ddreturn;
    static RECT rect = { 0, 0, 200, 190 };
    RECT DestRect, BufferRect;
    static int scale = 0;
    static int delta = 2;
    static DWORD TickCount = GetTickCount();

    //simple timing
    if ( (GetTickCount() - TickCount) < 50)
        return TRUE;
    TickCount = GetTickCount();

    //Restore back buffer pixels from previous frame
    do
    {
        ddreturn = lpBackbuffer->BltFast( (640/2)-(200/2), (480/2)-(190/2),
            lpOffBuffer, &rect,
            DDBLTFAST_NOCOLORKEY|DDBLTFAST_WAIT );
        if ( ddreturn == DDERR_SURFACELOST )
        {
            if ( !RestoreSurfaces() )
                return FALSE;
            //case - after restore, buffer is invalid
            BufferRect.left = (640/2)-(200/2);
            BufferRect.top = (480/2)-(190/2);
            BufferRect.right = BufferRect.left+200;
            BufferRect.bottom = BufferRect.top+190;
            ddreturn = lpOffBuffer->BltFast( 0, 0,
                lpBackbuffer, &BufferRect,
                DDBLTFAST_NOCOLORKEY|DDBLTFAST_WAIT );
            if ( ddreturn != DD_OK )
                return FALSE;
            break;
        }
    }
}

```

```

}while ( ddreturn != DD_OK );

//set up a destination rect
DestRect.left = (640/2)-((200-scale)/2);
DestRect.top = (480/2)-((190-scale)/2);
DestRect.right = DestRect.left + 200 - scale;
DestRect.bottom = DestRect.top + 190 - scale;

//Blt the offscreen surface to the back buffer
do
{
    ddreturn = lpBackbuffer->Blt( &DestRect,
        lpOffscreen, &rect, DDBLT_KEYSRC|DDBLT_WAIT, NULL );
    if ( ddreturn == DDERR_SURFACELOST )
    {
        if ( !RestoreSurfaces() )
            return FALSE;
    }
}while ( ddreturn != DD_OK );

//change the scale value for the next frame
scale += delta;
if (scale == 180 || scale == 0 )
    delta = -delta;

//Let DirectDraw switch our surface pointers
do
{
    ddreturn = lpPrimary->Flip ( NULL, DDFLIP_WAIT );
    if ( ddreturn == DDERR_SURFACELOST )
    {
        if ( !RestoreSurfaces() )
            return FALSE;
    }
}while ( ddreturn != DD_OK );

return TRUE;

} //DrawFrame

//*****
//RestoreSurfaces - restores all lost surfaces
//
//returns:
//    BOOL True - success
//*****
BOOL RestoreSurfaces ( void )
{
    HRESULT ddreturn;

```



```

        ddreturn = lpPrimary->Restore();
        if ( ddreturn == DD_OK )
        {
            if ( !BMPToDirectDrawSurface (lpPrimary, "back.bmp" ) )
                return FALSE;
        }
        else
        {
            return FALSE;
        }
        ddreturn = lpBackbuffer->Restore();
        if( ddreturn == DD_OK )
        {
            if ( !BMPToDirectDrawSurface (lpBackbuffer, "back.bmp" ) )
                return FALSE;
        }
        else
        {
            return FALSE;
        }
        }

        ddreturn = lpOffscreen->Restore();
        if( ddreturn == DD_OK )
        {
            if ( !BMPToDirectDrawSurface (lpOffscreen, "star.bmp" ) )
                return FALSE;
        }
        else
        {
            return FALSE;
        }
        }

        ddreturn = lpOffBuffer->Restore();
        if( ddreturn != DD_OK )
            return FALSE;

        return TRUE;

} //RestoreSurfaces

/*****
//BMPToDirectDrawSurface - opens and copies a BMP to a DD surface
//input:
//  LPDIRECTDRAWSURFACE surface
//  char *bmp - filename
//returns:
//  BOOL TRUE - success
*****/
BOOL BMPToDirectDrawSurface( LPDIRECTDRAWSURFACE surface, char *bmp )
{
    HRESULT          ddreturn;
    DWORD            actualRead;

```

```

HANDLE          hfile;
char            path[MAX_PATH];
BITMAPFILEHEADER BMPFileHead;
BITMAPINFOHEADER BMPFileInfo;
RGBQUAD         Palette[256];
PALETTEENTRY    pe[256];
LPDIRECTDRAWPALETTE lpDDPalette;
DDSURFACEDESC   ddsd;
int             i,src_width;
BYTE            *lpDDMemory,*lpBMPMemory,*image;

hfile = CreateFile( bmp, GENERIC_READ, FILE_SHARE_READ,
                    (LPSECURITY_ATTRIBUTES)NULL,
                    OPEN_EXISTING, FILE_ATTRIBUTE_NORMAL,
                    (HANDLE) NULL );

if ( hfile == INVALID_HANDLE_VALUE )
{
    strcpy (path,"..\");           //back up one directory
    strcat( path, bmp );
    hfile = CreateFile( path, GENERIC_READ, FILE_SHARE_READ,
                        (LPSECURITY_ATTRIBUTES)NULL,
                        OPEN_EXISTING, FILE_ATTRIBUTE_NORMAL,
                        (HANDLE) NULL );
    if ( hfile == INVALID_HANDLE_VALUE )
        return FALSE;
}

// Read the BMP header and info structures
if( !ReadFile ( hfile, &BMPFileHead, sizeof ( BMPFileHead ),
               &actualRead, NULL ) )
    return FALSE;

if( !ReadFile ( hfile, &BMPFileInfo, sizeof ( BMPFileInfo ),
               &actualRead, NULL ) )
    return FALSE;

//Make sure the BMP is 8-bit color
if ( BMPFileInfo.biBitCount != 8 )
    return FALSE;

//get the BMP palette
if ( !ReadFile ( hfile, Palette, sizeof( Palette ), &actualRead, NULL )
)
    return FALSE;

//Allocate memory for image data
image = (BYTE *) LocalAlloc ( LPTR,
                             BMPFileInfo.biWidth * BMPFileInfo.biHeight );
if( image == NULL )

```

```

return FALSE;

//read BMP into memory
if( !ReadFile ( hfile, image,
                BMPFileInfo.biWidth * BMPFileInfo.biHeight,
                &actualRead, NULL) )
{
    LocalFree( image );
    return FALSE;
}

//copy BMP data to DirectDraw surface
memset ( &ddsd, 0, sizeof( DDSURFACEDESC ) );
ddsd.dwSize = sizeof( ddsd );
ddreturn = surface->Lock( NULL, &ddsd, 0, NULL );

if( ddreturn != DD_OK )
{
    LocalFree( image );
    return FALSE;
}

lpDDMemory = (BYTE *) ddsd.lpSurface;
lpBMPMemory = image+((BMPFileInfo.biHeight-1)*BMPFileInfo.biWidth);

if ( BMPFileInfo.biWidth > ddsd.lPitch )
    src_width = ddsd.lPitch;
else
    src_width = BMPFileInfo.biWidth;

for( i = 0; i < BMPFileInfo.biHeight; i++ )
{
    memcpy ( lpDDMemory, lpBMPMemory, src_width );
    lpDDMemory += ddsd.lPitch;
    lpBMPMemory -= src_width;
}

surface->Unlock ( NULL );
LocalFree( image );
CloseHandle(hfile);

//create a DirectDraw palette for the surface
for( i=0; i<256; i++ )
{
    pe[i].peRed = Palette[i].rgbRed;
    pe[i].peGreen = Palette[i].rgbGreen;
    pe[i].peBlue = Palette[i].rgbBlue;
}
ddreturn = lpDirectDrawObject->CreatePalette( DDPCAPS_8BIT,
                                              pe,

```

```

                                                                    &lpDDPalette,
                                                                    NULL );
if ( ddreturn != DD_OK )
    return FALSE;
surface->SetPalette ( lpDDPalette );

    return TRUE;
} //BMPToDirectDrawSurface
```

Blting: DirectDraw Clippers

Graphics clipping is a simple technique that defines areas of a drawing surface where it is legal to write pixels, and conversely, areas within or outside of the drawing surface that are off limits.

In this chapter, we will explore DirectDraw's implementation of clipping using a **DirectDrawClipper** object and write a program that uses clipping to produce a scrolling effect.

CLIPPING

The most important use of clipping is to ensure that blted images are never written outside of the legal range of the destination drawing surface. For example, if a source image of 100×100 is to be blted to a 640×480 surface, starting at location 600, 100 of the destination surface, then the source image must be clipped along the x axis or the image will be blted to a section of memory that is not a part of the destination surface, resulting in a general protection fault. Figure 10.1 illustrates a clip based on the size of the destination surface.

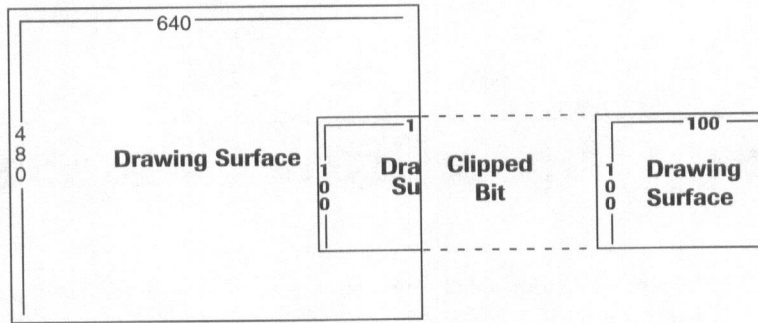


Figure 10.1 A clip based on the size of the destination surface.

A second type of clipping involves setting up zones within the destination surface where source images can be blt'd. If any part of the source image falls outside of the legal zones, it is clipped, and any pixel data on the destination surface outside of the blt zones is preserved. Figure 10.2 provides an example of this type of clipping.

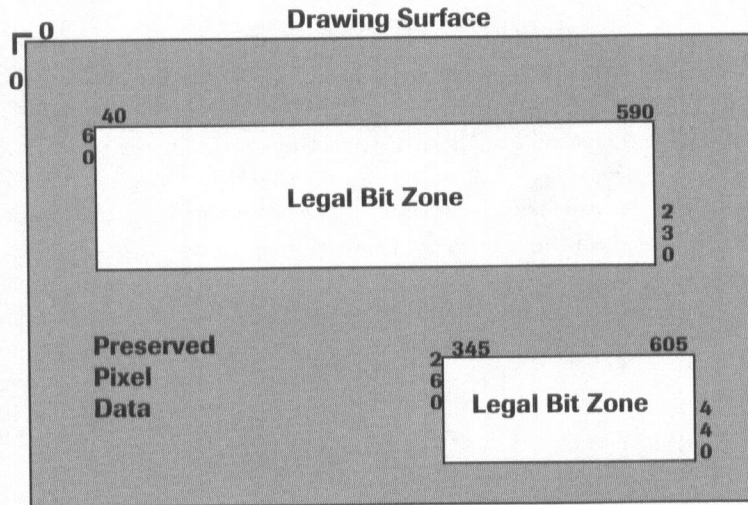


Figure 10.2 Clipping using zones.

Both types of clipping are useful and are often combined. Clipping along the legal range of the destination surface is mandatory when source images must be blitted along the edges of the surface, in a platform scrolling game, for example. Clipping within legal zones is used to create windows within the destination surface. One window, or clip zone might contain the primary action for the game; an enemy-populated star field in a 3D shooter comes to mind. While another window within the destination surface might be used to indicate the status of the player's ship; shield strength, amount of available ammunition, and so on. Using clipping, we can ensure that the 3D star field is never written to the section of the destination surface that contains the ship status information and vice versa.

CLIPPING UNDER DIRECTDRAW

DirectDraw provides a high-level DirectDraw object, the **DirectDrawClipper**, that can be used for destination clipping. **DirectDrawClipper** objects can contain multiple clip lists (the rectangles that make up the clip zones), can be attached to multiple surfaces, and most importantly, the **DirectDrawClipper** object provides an elegant mechanism for clipping graphics in a "real" window (i.e., a windowed program that looks and behaves like any other Windows application).

THE CREATECLIPPER FUNCTION

The **CreateClipper** function is used to create a **DirectDrawClipper** object:

```
HRESULT CreateClipper(
    LPDIRECTDRAW lpDD,
    DWORD dwFlags,
    LPDIRECTDRAWCLIPPER FAR *lpDDClipper,
    IUnknown FAR *pUnkOuter )
```

The first parameter, **lpDD**, points to the DirectDraw object that was instantiated with a call to **DirectDrawCreate**. The **dwFlags** parameter is currently not used and should be set to null. **lpDDClipper** points to the pointer that will receive the address of the newly created **DirectDrawClipper** object. The last parameter,

pUnkOuter, is a part of the Microsoft COM interface and should be set to **NULL**. A return of **DD_OK** indicates that the **DirectDrawClipper** object was successfully created. A complete list of error returns is found in Appendix A.

CREATING CLIP LISTS

Once the **DirectDrawClipper** object is created, we can decide on the type and number of clip lists that the **DirectDrawClipper** object will contain. Multiple clip lists can be created by calling the **SetClipList** member function, discussed shortly. A single clip list, sized at the dimensions of the current window, can be created by calling the **SetHWND** member function.

THE SETHWND FUNCTION

The **SetHWND** function creates a clip list that is tied to the application's window size. At the start of the DirectDraw application, this clip list will equal the dimensions of the window as expressed in the **CreateWindowEx** API function. If the DirectDraw program is running in full-screen mode, this clip list will not change because the window size is constant. On the other hand, if the DirectDraw program is running in a Windows window, then the clip list created by **SetHWND** will be updated by DirectDraw, as necessary, to reflect any changes in the window's size or position that the user or the program itself may have altered. Running DirectDraw programs in a window will be covered in detail in Chapter 12.

```
HRESULT SetHWND(
    LPDIRECTDRAWCLIPPER lpDDClipper,
    DWORD dwFlags,
    HWND hWnd )
```

lpDDClipper points to a **DirectDrawClipper** object. The **dwFlags** parameter is for future expansion and should be set to **NULL**. The **hWnd** parameter is the handle of the window that **SetHWND** will use to generate the new clip list. A return of **DD_OK** indicates success.

THE SETCLIPLIST FUNCTION

The **SetHWnd** function presents a compact, convenient method for creating a clip list. A limitation of the **SetHWnd** function is that only one clip list per **DirectDrawClipper** object can be maintained, and because the clip list created by **SetHWnd** is tied to the changeable size and positional data of a window, it isn't a good candidate for producing multiple clip lists that will be applied to a single drawing surface. We can apply multiple clip lists to a single **DirectDrawClipper** object by using the **SetClipList** member function.

```
HRESULT SetClipList(  
    LPDIRECTDRAWCLIPPER lpDDClipper,  
    LPRGNDATA lpClipList,  
    DWORD dwFlags )
```

lpDDClipper points to a **DirectDrawClipper** object. The second parameter, **lpClipList**, either points to a **RGNDATA** (discussed shortly) structure, or, if **lpClipList** is set to **NULL**, any clip list that is already attached to this **DirectDrawClipper** object will be removed. **dwFlags** is not currently used and can be set to **NULL**. A return of **DD_OK** indicates success.

USING THE RGNDATA STRUCTURE

A **RGNDATA** structure contains an arbitrary number of rectangles (**RECT** structures) that make up a region. This region, in the context of the **DirectDrawClipper** object, defines one or more clip lists that can be attached to a drawing surface. The Microsoft Win32 API help guide defines the **RGNDATA** structure as:

```
typedef struct _RGNDATA {  
    RGNDATAHEADER rdh;  
    char Buffer [1];  
} RGNDATA;
```

This type of structure prototyping can be confusing. The problem lies in the **Buffer** declaration, included to suppress warning messages from the C compiler.

What will actually be placed at **Buffer**'s location within **RGNDATA** is a variable number of **RECT** structures. The characteristics of the **RGNDATA** structure are defined by the **RGNDATAHEADER** substructure:

```
typedef struct _RGNDATAHEADER {
    DWORD dwSize;
    DWORD iType;
    DWORD nCount;
    DWORD nRgnSize;
    RECT rcBound;
} RGNDATAHEADER;
```

The **dwSize** field is the size of the **RGNDATAHEADER** structure, in bytes. The **iType** field specifies the type of region data and should be set to **RDH_RECTANGLES**. The **nCount** field is the number of **RECT**s that make up the region data. The **nRgnSize** field defines the size of the variable-length **RECT**s, in bytes. The value in this field can be computed as:

```
sizeof (RECT) * NUMBER_OF_RECTANGLES
```

The **rcBound** field is a **RECT** structure that defines a bounding rectangle for the entire region. The **rcBound** field can be used to define the legal limits of a drawing surface.

A SETCLIPLIST EXAMPLE

Using Figure 10.2 as an example, we can create a multiple clip list with the following source code fragment:

```
LPDIRECTDRAWCLIPPER lpDDClipper;

typedef struct
{
    RGNDATAHEADER rdh;
    RECT MainWnd;
    RECT StatusWnd;
} ClipLists;

//Build the header
```

```
ClipLists.rdh.dwSize = sizeof (RGNDATAHEADER);
ClipLists.rdh.iType = RDH_RECTANGLES;
ClipLists.rdh.nCount = 2;
ClipLists.rdh.nRegionSize = sizeof (RECT) * 2;
//Define the legal limits of the surface
ClipLists.rdh.rcBound.left = 0;
ClipLists.rdh.rcBound.top = 0;
ClipLists.rdh.rcBound.right = 640;
ClipLists.rdh.rcBound.bottom = 480;
//Define an interior clip list
ClipLists.MainWnd.rcBound.left = 40;
ClipLists.MainWnd.rcBound.top = 60;
ClipLists.MainWnd.rcBound.right = 590;
ClipLists.MainWnd.rcBound.bottom = 230
//Define an interior clip list
ClipLists.StatusWnd.rcBound.left = 345;
ClipLists.StatusWnd.rcBound.top = 260;
ClipLists.StatusWnd.rcBound.right = 605;
ClipLists.StatusWnd.rcBound.bottom = 440;
//Assumes lpDDClipper points to a DirectDraw clipper object
SetClipList ( lpDDClipper, (RGNDATA *) &ClipLists, 0 );
```

THE SETCLIPPER FUNCTION

Once a clip list has been created and applied to a **DirectDrawClipper** object, either through **SetHWND** or **SetClipList**, it can be attached to a drawing surface by calling the **SetClipper** function.

```
HRESULT SetClipper(
    LPDIRECTDRAWSURFACE lpDDSurface,
    LPDIRECTDRAWCLIPPER lpDDClipper)
```

lpDDSurface points to the surface that will receive the **DirectDrawClipper** object. **lpDDClipper** points to the clipper object that should be attached to **lpSurface**. This parameter can also be NULL. In this case, any clipper objects that are attached to the surface pointed at by **lpDDSurface** will be removed.

CLIPPER LIMITATIONS

It is important to keep in mind the following limitations when using **DirectDrawClipper** objects:

1. The **BltFast** function cannot be used on a surface with an attached **DirectDrawClipper** object. These surfaces must use **Blt**.
2. Clip lists created with **SetHWnd** and those created with **SetClipList** cannot be combined. The programmer must choose between the two DirectDraw clipper methods.
3. When creating a clip list using the **SetHWnd** method, the speed penalty is small and more than offset by the ease of use of this function. On the other hand, multiple clip lists created with **SetClipList** should be used with care, as a substantial drop in blting speed can be induced. When multiple clip lists are required within a single drawing surface, it is probably more efficient to write a dedicated clipping function.

THE CLIP.CPP PROGRAM

The following program creates and attaches a **DirectDrawClipper** object, using **SetHWnd**, to a back buffer surface. Unlike the previous demonstration programs, CLIP.CPP buffers entire the back buffer surface and requires a video card with more than 1 MB of RAM.

```
//*****
//DirectDraw Programming
//Bret Timmins
//
//(C)1995 M&T Books
//
//CLIP.CPP
//  Demonstrates clipping using SetHWnd
//*****

#define WIN32_LEAN_AND_MEAN
#include <windows.h>
#include <windowsx.h>
#include <ddraw.h>
#include <stdio.h>
#include <stdlib.h>
#include <stdarg.h>

#define SCREEN_WIDTH 640
#define SCREEN_HEIGHT 480

#define SURFACE_WIDTH 516
#define SURFACE_HEIGHT 204
```

DIRECTDRAW PROGRAMMING

Chapter Ten: Blitting: DirectDraw Clippers

```
LPDIRECTDRAW      lpDirectDrawObject;    // DirectDraw object
LPDIRECTDRAWSURFACE lpPrimary;           // DirectDraw primary
surface
LPDIRECTDRAWSURFACE lpBackbuffer;        //DirectDraw backbuffer
surface
LPDIRECTDRAWSURFACE lpOffscreen;         // Offscreen surface image
LPDIRECTDRAWSURFACE lpOffbuffer;         // Offscreen surface buffer
// entire display surface
LPDIRECTDRAWCLIPPER lpClipper;           // Clipper object
BOOL              ActiveApp;            // Is this program active?

int PASCAL WinMain( HINSTANCE hInstance, HINSTANCE hPrevInstance,
                    LPSTR lpCmdLine, int nCmdShow);
long FAR PASCAL WindowProc ( HWND hwnd, UINT message,
                             WPARAM wParam, LPARAM lParam );

BOOL DrawFrame ( void );
BOOL RestoreSurfaces ( void );
BOOL BMPToDirectDrawSurface( LPDIRECTDRAWSURFACE surface, char *bmp );

//*****
//WinMain - mandatory windows init function
//
//*****
int PASCAL WinMain( HINSTANCE hInstance, HINSTANCE hPrevInstance,
                    LPSTR lpCmdLine, int nCmdShow)
{
    MSG          msg;
    HWND         hwnd;
    WNDCLASS     wc;
    static char ClassName[] = "CLIP";
    DDSURFACEDESC ddsd;
    DDSCAPS      ddscaps;
    DDCOLORKEY   ddck;
    HRESULT      ddreturn;

    lpCmdLine = lpCmdLine;
    hPrevInstance = hPrevInstance;

    //register and realize our display window
    wc.style = CS_HREDRAW | CS_VREDRAW;
    wc.lpfnWndProc = WindowProc;
    wc.cbClsExtra = 0;
    wc.cbWndExtra = 0;
    wc.hInstance = hInstance;
    wc.hIcon = LoadIcon( hInstance, IDI_APPLICATION );
    wc.hCursor = LoadCursor( NULL, IDC_ARROW );
    wc.hbrBackground = NULL;
    wc.lpszMenuName = ClassName;
    wc.lpszClassName = ClassName;
    RegisterClass( &wc );
```

```
//create a full screen window so that GDI won't ever be called
hwnd = CreateWindowEx(
    WS_EX_TOPMOST,
    ClassName,
    ClassName,
    WS_POPUP,
    0,
    0,
    GetSystemMetrics(SM_CXSCREEN),
    GetSystemMetrics(SM_CYSCREEN),
    NULL,
    NULL,
    hInstance,
    NULL );

if( !hwnd )
    return FALSE;

ShowWindow( hwnd, nCmdShow );
UpdateWindow( hwnd );
SetFocus( hwnd );
ShowCursor( FALSE );

//Instanciate our DirectDraw object
ddreturn = DirectDrawCreate( NULL, &lpDirectDrawObject, NULL );
if ( ddreturn != DD_OK )
{
    DestroyWindow( hwnd );
    return FALSE;
}

ddreturn = lpDirectDrawObject->SetCooperativeLevel( hwnd,
    DDSCL_EXCLUSIVE | DDSCL_FULLSCREEN );
if ( ddreturn != DD_OK )
{
    DestroyWindow( hwnd );
    return FALSE;
}

// Set the video mode to 640x480x8
ddreturn=lpDirectDrawObject->SetDisplayMode(SCREEN_WIDTH,
SCREEN_HEIGHT,8);
if ( ddreturn != DD_OK )
{
    DestroyWindow( hwnd );
    return FALSE;
}

// Create the primary surface and one back buffer surface
ddsd.dwSize = sizeof( ddsd );
```

```

ddsd.dwFlags = DDSD_CAPS | DDSD_BACKBUFFERCOUNT;
ddsd.ddsCaps.dwCaps = DDSCAPS_PRIMARYSURFACE |
                      DDSCAPS_FLIP |
                      DDSCAPS_COMPLEX;
ddsd.dwBackBufferCount = 1;
ddreturn = lpDirectDrawObject->CreateSurface( &ddsd, &lpPrimary, NULL
);
    if ( ddreturn != DD_OK )
    {
        DestroyWindow( hwnd );
        return FALSE;
    }

//get a surface pointer to our back buffer
ddscaps.dwCaps = DDSCAPS_BACKBUFFER;
ddreturn = lpPrimary->GetAttachedSurface ( &ddscaps, &lpBackbuffer);
if( ddreturn != DD_OK )
{
    DestroyWindow ( hwnd );
    return FALSE;
}

// Create a clipper
ddreturn = lpDirectDrawObject->CreateClipper ( 0, &lpClipper, NULL );
if ( ddreturn != DD_OK )
{
    DestroyWindow( hwnd );
    return FALSE;
}

// Get coordinates of our window
ddreturn = lpClipper->SetHWND ( 0, hwnd );
if ( ddreturn != DD_OK )
{
    DestroyWindow( hwnd );
    return FALSE;
}

// Apply Clipper to the back buffer
ddreturn = lpBackbuffer->SetClipper( lpClipper );
if ( ddreturn != DD_OK )
{
    DestroyWindow( hwnd );
    return FALSE;
}

// Create a buffer for display memory
ddsd.dwFlags = DDSD_CAPS | DDSD_HEIGHT | DDSD_WIDTH;
ddsd.ddsCaps.dwCaps = DDSCAPS_OFFSCREENPLAIN;

```

```
ddsd.dwHeight = SCREEN_HEIGHT;
ddsd.dwWidth = SCREEN_WIDTH;
ddreturn=lpDirectDrawObject->CreateSurface ( &ddsd, &lpOffbuffer, NULL
);
    if ( ddreturn != DD_OK )
    {
        DestroyWindow( hwnd );
        return FALSE;
    }

    // Create the main offscreen surface
    ddsd.dwFlags = DDSD_CAPS | DDSD_HEIGHT | DDSD_WIDTH;
    ddsd.ddsCaps.dwCaps = DDSCAPS_OFFSCREENPLAIN;
    ddsd.dwHeight = SURFACE_HEIGHT;
    ddsd.dwWidth = SURFACE_WIDTH;
    ddreturn = lpDirectDrawObject->CreateSurface (&ddsd, &lpOffscreen, NULL
);
    if ( ddreturn != DD_OK )
    {
        DestroyWindow( hwnd );
        return FALSE;
    }

    //Set the transparent color key for the off screen surface
    ddck.dwColorSpaceLowValue = 255;
    ddck.dwColorSpaceHighValue = 255;
    lpOffscreen->SetColorKey( DDCKEY_SRCBLT, &ddck );

    //copy a BMP file to the primary surface
    if ( !BMPToDirectDrawSurface (lpPrimary, "desert.bmp" ) )
    {
        DestroyWindow ( hwnd );
        return FALSE;
    }

    //copy to the back buffer
    if ( !BMPToDirectDrawSurface (lpBackbuffer, "desert.bmp" ) )
    {
        DestroyWindow ( hwnd );
        return FALSE;
    }

    //copy to the display buffer
    if ( !BMPToDirectDrawSurface (lpOffbuffer, "desert.bmp" ) )
    {
        DestroyWindow ( hwnd );
        return FALSE;
    }

    //copy to the offscreen surface
```



```
if ( !BMPToDirectDrawSurface (lpOffscreen, "arizona.bmp" ) )
{
    DestroyWindow ( hwnd );
    return FALSE;
}

while( 1 )
{
    if( PeekMessage( &msg, NULL, 0, 0, PM_NOREMOVE ) )
    {
        if( !GetMessage( &msg, NULL, 0, 0 ) )
            return msg.wParam;
        TranslateMessage(&msg);
        DispatchMessage(&msg);
    }else if ( ActiveApp )
    {
        DrawFrame ();
    }else
    {
        WaitMessage ();
    }
}

} //WinMain

//*****
//WindowProc - receive and handle windows messages
//
//*****
long FAR PASCAL WindowProc( HWND hwnd, UINT message,
                           WPARAM wParam, LPARAM lParam )
{
    switch( message )
    {
        case WM_ACTIVATEAPP:
            ActiveApp = wParam;
            break;

        case WM_CREATE:
            break;

        case WM_KEYDOWN:
            switch ( wParam )
            {
                case VK_ESCAPE:
                    DestroyWindow ( hwnd );
                    break;
            }
            break;
    }
}
```

```

case WM_DESTROY:
    if ( lpDirectDrawObject != NULL )
    {
        if ( lpOffscreen != NULL )
            lpOffscreen->Release();
        if ( lpOffbuffer != NULL )
            lpOffbuffer->Release();
        if ( lpPrimary != NULL )
            lpPrimary->Release();
        lpDirectDrawObject->Release();
    }

    ShowCursor ( TRUE );
    PostQuitMessage ( wParam );
    break;

default:
    return DefWindowProc ( hwnd, message, wParam, lParam );
}

return 0L;

} //WindowProc

//*****
//DrawFrame - Compose a frame of graphics, flip surfaces
//
//return:
//  BOOL TRUE - success
//*****
BOOL DrawFrame ( void )
{
    HRESULT ddreturn;
    static RECT ImageRect = { 0, 0, SURFACE_WIDTH,SURFACE_HEIGHT };
    static RECT BufferRect = { 0,0,SCREEN_WIDTH,SCREEN_HEIGHT };
    RECT AnimRect;
    static int XPos = 100, YPos = -SURFACE_HEIGHT;

    //clean up the back buffer from the previous frame by blting
    //the pixel data on the display buffer to the back buffer
    do
    {
        ddreturn = lpBackbuffer->Blt( &BufferRect,
                                     lpOffbuffer, &BufferRect,
                                     DDBLT_WAIT,NULL );
        if ( ddreturn == DDERR_SURFACELOST )
        {
            if ( !RestoreSurfaces() )
                return FALSE;
        }
    }

```

```

}while ( ddreturn != DD_OK );

//Blt the offscreen surface to the back buffer
AnimRect.left = XPos;
AnimRect.top = YPos;
AnimRect.right = XPos+SURFACE_WIDTH;
AnimRect.bottom = YPos+SURFACE_HEIGHT;
do
{
    ddreturn = lpBackbuffer->Blt( &AnimRect,
                                lpOffscreen,&ImageRect,DDBLT_KEYSRC|DDBLT_WAIT,NULL
);
    if ( ddreturn == DDERR_SURFACELOST )
    {
        if ( !RestoreSurfaces() )
            return FALSE;
    }
}while ( ddreturn != DD_OK );

YPos++;
if ( YPos > SCREEN_HEIGHT )
    YPos = -SURFACE_HEIGHT;

//Let DirectDraw switch our surface pointers
do
{
    ddreturn = lpPrimary->Flip ( NULL, DDFLIP_WAIT );
    if ( ddreturn == DDERR_SURFACELOST )
    {
        if ( !RestoreSurfaces() )
            return FALSE;
    }
}while ( ddreturn != DD_OK );

return TRUE;

} //DrawFrame

//*****
//RestoreSurfaces - restores all lost surfaces
//
//returns:
//  BOOL True - success
//*****
BOOL RestoreSurfaces ( void )
{
    HRESULT ddreturn;

    ddreturn = lpPrimary->Restore();
    if ( ddreturn == DD_OK )

```

```

{
    if ( !BMPToDirectDrawSurface (lpPrimary, "desert.bmp" ) )
        return FALSE;
}
else
{
    return FALSE;
}
ddreturn = lpBackbuffer->Restore();
if( ddreturn == DD_OK )
{
    if ( !BMPToDirectDrawSurface (lpBackbuffer, "desert.bmp" ) )
        return FALSE;
}
else
{
    return FALSE;
}

ddreturn = lpOffbuffer->Restore();
if( ddreturn == DD_OK )
{
    if ( !BMPToDirectDrawSurface (lpOffbuffer, "desert.bmp" ) )
        return FALSE;
}
else
{
    return FALSE;
}

ddreturn = lpOffscreen->Restore();
if( ddreturn == DD_OK )
{
    if ( !BMPToDirectDrawSurface (lpOffscreen, "arizona.bmp" ) )
        return FALSE;
}
else
{
    return FALSE;
}

return TRUE;
}

//RestoreSurfaces

//*****
//BMPToDirectDrawSurface - opens and copies a BMP to a DD surface
//input:
//    LPDIRECTDRAWSURFACE surface
//    char *bmp - filename
//returns:
//    BOOL TRUE - success
//*****

```

```

BOOL BMPToDirectDrawSurface( LPDIRECTDRAWSURFACE surface, char *bmp )
{
    HRESULT                ddreturn;
    DWORD                  actualRead;
    HANDLE                 hfile;
    char                   path[MAX_PATH];
    BITMAPFILEHEADER       BMPFileHead;
    BITMAPINFOHEADER       BMPFileInfo;
    RGBQUAD                 Palette[256];
    PALETTEENTRY            pe[256];
    LPDIRECTDRAWPALETTE     lpDDPalette;
    DDSURFCEDESC            ddsd;
    int                    i,src_width;
    BYTE                   *lpDDMemory,*lpBMPMemory,*image;

    hfile = CreateFile( bmp, GENERIC_READ, FILE_SHARE_READ,
                      (LPSECURITY_ATTRIBUTES)NULL,
                      OPEN_EXISTING, FILE_ATTRIBUTE_NORMAL,
                      (HANDLE) NULL );

    if ( hfile == INVALID_HANDLE_VALUE )
    {
        strcpy( path,"..\\");
        directory
        strcat( path, bmp );
        hfile = CreateFile( path, GENERIC_READ, FILE_SHARE_READ,
                          (LPSECURITY_ATTRIBUTES)NULL,
                          OPEN_EXISTING, FILE_ATTRIBUTE_NORMAL,
                          (HANDLE) NULL );
        if ( hfile == INVALID_HANDLE_VALUE )
            return FALSE;
    }

    // Read the BMP header and info structures
    if( !ReadFile( hfile, &BMPFileHead, sizeof( BMPFileHead ),
                  &actualRead, NULL ) )
        return FALSE;

    if( !ReadFile( hfile, &BMPFileInfo, sizeof( BMPFileInfo ),
                  &actualRead, NULL ) )
        return FALSE;

    //Make sure the BMP is 8-bit color
    if ( BMPFileInfo.biBitCount != 8 )
        return FALSE;

    //get the BMP palette
    if ( !ReadFile( hfile, Palette, sizeof( Palette ), &actualRead, NULL )
)

```

```
    return FALSE;

    //Allocate memory for image data
    image = (BYTE *) LocalAlloc ( LPTR,
                                   BMPFileInfo.biWidth * BMPFileInfo.biHeight );
    if( image == NULL )
        return FALSE;

    //read BMP into memory
    if( !ReadFile ( hfile, image,
                    BMPFileInfo.biWidth * BMPFileInfo.biHeight,
                    &actualRead, NULL) )
    {
        LocalFree( image );
        return FALSE;
    }

    //copy BMP data to DirectDraw surface
    memset ( &ddsd, 0, sizeof( DDSURFACEDESC ) );
    ddsd.dwSize = sizeof( ddsd );
    ddreturn = surface->Lock( NULL, &ddsd, 0, NULL );

    if( ddreturn != DD_OK )
    {
        LocalFree( image );
        return FALSE;
    }

    lpDDMemory = (BYTE *) ddsd.lpSurface;
    lpBMPMemory = image+((BMPFileInfo.biHeight-1)*BMPFileInfo.biWidth);

    if ( BMPFileInfo.biWidth > ddsd.lPitch )
        src_width = ddsd.lPitch;
    else
        src_width = BMPFileInfo.biWidth;

    for( i = 0; i < BMPFileInfo.biHeight; i++ )
    {
        memcpy ( lpDDMemory, lpBMPMemory, src_width );
        lpDDMemory += ddsd.lPitch;
        lpBMPMemory -= src_width;
    }

    surface->Unlock ( NULL );
    LocalFree( image );
    CloseHandle(hfile);

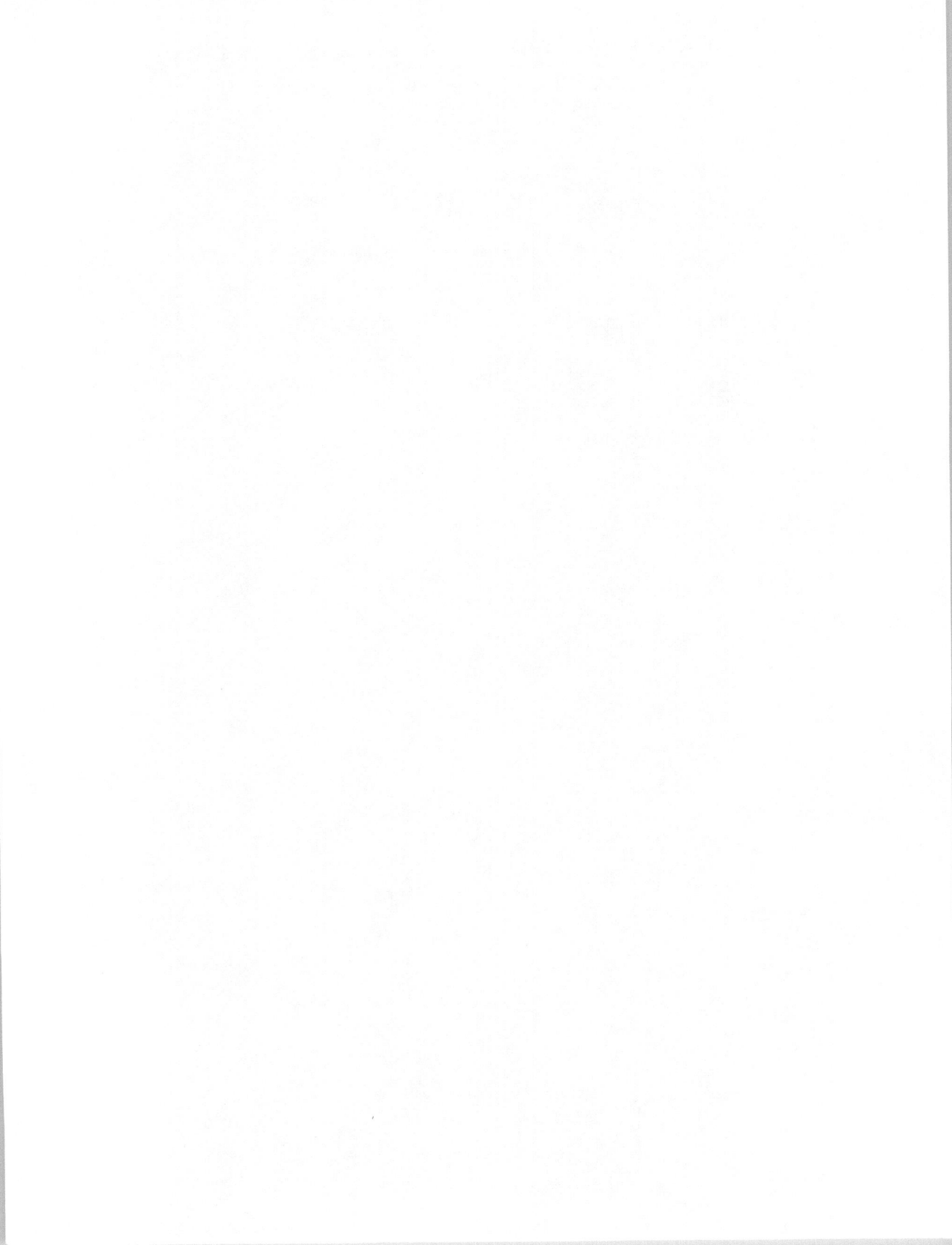
    //create a DirectDraw palette for the surface
    for( i=0; i<256; i++ )
    {
```

```
    pe[i].peRed = Palette[i].rgbRed;
    pe[i].peGreen = Palette[i].rgbGreen;
    pe[i].peBlue = Palette[i].rgbBlue;
}
ddreturn = lpDirectDrawObject->CreatePalette( DDPCAPS_8BIT,
                                              pe,
                                              &lpDDPalette,
                                              NULL );

if ( ddreturn != DD_OK )
    return FALSE;
surface->SetPalette ( lpDDPalette );

return TRUE;

} //BMPToDirectDrawSurface
```



DirectDraw Color

One of DirectDraw's primary design goals is to support a broad range of video modes and color depths. DirectDraw can handle palletized RGB color, palette-independent RGB modes, and non-RGB modes.

From the programmer's standpoint, DirectDraw offers a degree of abstraction from the operational details of various color modes (i.e., the **Blt** and **BltFast** functions will operate at all supported color depths without any special intervention or setup by the programmer). But this high-level approach can't be used when dealing with drawing surfaces at the pixel level (e.g., when initializing a surface with image data). In these cases, we can use the color information maintained by DirectDraw to correctly access drawing surface pixels.

In this chapter, we will document DirectDraw's implementation of color and write a program that displays a high-color image.

PALETTE MODES

VGA and SVGA color palette modes create color using an indirect indexing system. The indices are the pixel values within the image, and the elements that are indexed are an internal set of palette registers within the SVGA card. A pixel with a value of zero, for example, will derive its color from the first of the internal palette registers. A pixel value of 255 will reference the last palette register (in a 256 color video mode), and so on. The value within an internal palette register determines the actual color that is displayed at a given pixel's position. Figure 11.1 illustrates the palette indexing system.

At the chip level, it is technically incorrect to refer to a single palette register as containing R, G, and B values. SVGA cards actually contain 768 palette registers, with each register containing a percentage of red, green, or blue. Figure 11.2 illustrates the organization of these registers.

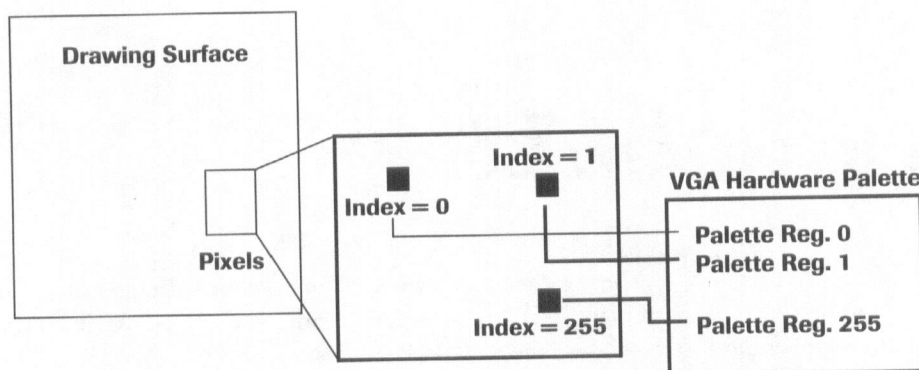


Figure 11.1 The palette indexing system.

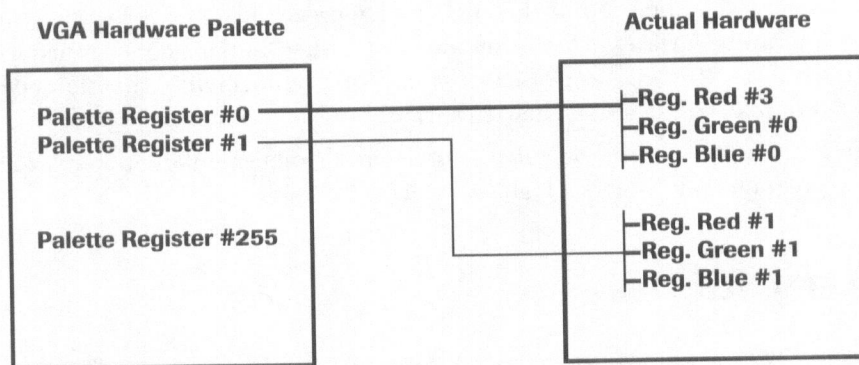


Figure 11.2 Organization of palette registers.

PALETTES AND DIRECTDRAW

While DirectDraw doesn't allow low-level access to the palette registers, it does provide functions to read and write to them through the `DirectDrawPalette` object, and it allows the programmer to maintain a greater degree of control over color usage than the Window's GDI palette functions when running in DirectDraw exclusive mode. Note that this situation is somewhat reversed when running in a window with DirectDraw's normal cooperation level. Running windowed will be covered in Chapter 12.

THE CREATEPALETTE FUNCTION

The first step in creating a DirectDraw palette is to create a `DirectDrawPalette` object. This is done through the DirectDraw object function **CreatePalette**:

```
HRESULT CreatePalette(
    LPDIRECTDRAW lpDD,
    DWORD dwFlags,
    LPPALETTEENTRY lpColorTable,
    LPDIRECTDRAWPALETTE FAR* lpDDPalette,
    IUnknown FAR *pUnkOuter)
```

The first parameter, **lpDD**, points to the DirectDraw object that was created with the **DirectDrawCreate** function. The **dwFlags** parameter controls the type of palette and how it should be initialized:

DDPCAPS_4BIT

This flag informs DirectDraw that the palette will define only 16 colors. Note that this flag doesn't denote an actual 16-color video mode, which DirectDraw doesn't currently support, but is used with the following **DDPCAPS_8BITENTRIES** flag:

DDPCAPS_8BITENTRIES

The **DDPCAPS_8BITENTRIES** flag, which must be combined with **DDCAPS_4BIT**, informs DirectDraw that the 16-color palette data (BYTE length) passed in **lpColorTable** (see following) should be treated as indexes into an existing palette. When this type of indirect palette is applied to a surface, the pixels of the surface are decoded in the following manner (Figure 11.3):

By applying an offset to the **DDCAPS_8BITENTRIES** starting index number, pixel data can be stored using 4 bits to represent the pixel instead of 8 bits.

DDPCAPS_8BIT

The **DDPCAPS_8BIT** flag denotes a standard 256-color palette:

DDPCAPS_ALLOW256

By default, DirectDraw will prevent palette register 0 and register 255 from being overwritten. The **DDCAPS_ALLOW256** flag allows the entire set of palette regis-

ters to be changed. In DirectDraw's default 254 color palette mode, indexes 0 and 255 are good candidates for transparent pixel values.

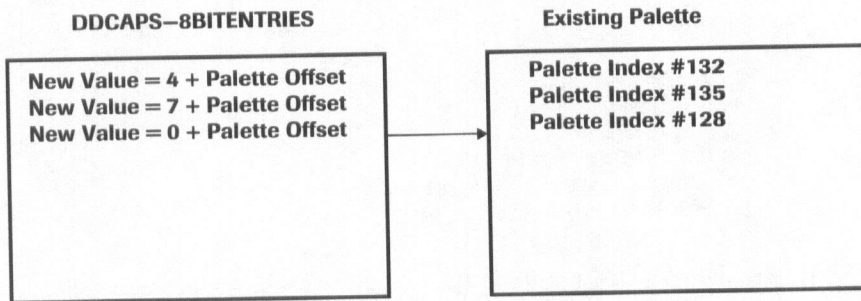


Figure 11.3 The base Palette Offset value equals 128.

The next parameter of **CreatePalette**, **lpColorTable**, points to the RGB color values that will make up the palette. The number of palette entries must be 16 or 256 (without the **DDPCAPS_ALLOW256** flag, entries 0 and 255 will be ignored). The palette data passed should be in the Windows API **PALETTEENTRY** format. **lpDDPalette** points to a pointer that will receive the address of the newly created **DirectDrawPalette** object. The **pUnkOuter** parameter is a part of the COM interface and should be set to **NULL**. A return of **DD_OK** indicates success.

THE SETPALETTE FUNCTION

Once a **DirectDrawPalette** object has been created, the object's palette can be applied to a drawing surface by calling the **SetPalette** member function:

```

HRESULT SetPalette(
    LPDIRECTDRAWSURFACE lpDDSurface,
    LPDIRECTDRAWPALETTE lpDDPalette)
    
```

lpDDSurface points to the surface object that will receive the palette. **lpDDPalette** points to the **DirectDrawPalette** object. A return of **DD_OK** indicates success.

THE SETENTRIES AND GETENTRIES FUNCTIONS

Once a palette has been attached to a surface with **SetPalette**, individual palette entries can be accessed with the **SetEntries** and **GetEntries** functions:

```
HRESULT SetEntries(  
    LPDIRECTDRAWPALETTE lpDDPalette,  
    DWORD dwFlags,  
    DWORD dwStartingEntry,  
    DWORD dwCount,  
    LPPALETTEENTRY lpEntries)
```

The first parameter, **lpDDPalette**, points the **DirectDrawPalette** object. that was created with **CreatePalette**. **dwFlags** is currently not used and should be set to **NULL**. The **dwStartingEntry** parameter is the first palette entry within the **DirectDraw** palette that will be changed. **dwCount** determines the number of palette entries, from **dwStartingEntry**, that will be overwritten. The **lpEntries** parameter points to the new palette data in the **PALETTEENTRY** format, or if the **DDPCAPS_8BITENTRIES** flag was specified when the palette was created, **lpEntries** should point to a **BYTE** length data field. A return of **DD_OK** indicates success.

The following source code fragment writes new color data to palette entries 16 through 32:

```
PALETTEENTRY palette[16];  
LPDIRECTDRAWPALETTE lpDirectDrawPalette;  
//Assume CreatePalette and SetPalette have been called  
//Fill palette array with new color data  
lpDirectDrawPalette ( 0, 16 ,16 , palette );
```

The **GetEntries** function returns palette information in a **PALETTEENTRY** structure:

```
HRESULT GetEntries(  
    LPDIRECTDRAWPALETTE lpDDPalette,  
    DWORD dwFlags,  
    DWORD dwBase,  
    DWORD dwNumEntries,  
    LPPALETTEENTRY lpEntries )
```

lpDDPalette points to the **DirectDrawPalette** object.. **dwFlags** is not used and should be set to **NULL**. **dwBase** is the first palette entry that should be returned

and **dwNumEntries** represents the number of palette entries, starting from **dwBase**, that will be read. **lpEntries** points to a **PALETTEENTRY** array or a **BYTE** length array if the **DDPCAPS_8BITENTRIES** flag was used during creation of the palette, that will receive the color information. A return of **DD_OK** indicates success.

SetEntries and the **GetEntries** functions are most useful for palette animation and fading routines.

PALETTE-INDEPENDENT COLOR

Hicolor and true color video modes are palette-independent in the sense that each pixel directly represents the color the user will see at the pixel's position. Hicolor modes have 15 or 16 bits of color data per pixel (bpp), while true color is represented with 24 or 32 bpp.



True color video modes are not supported in the current release of the Microsoft's Games SDK.

HICOLOR RGB VIDEO MODES

Hicolor video modes use a **WORD** quantity to represent the RGB values that will make up the color of a pixel. Within the pixel (**WORD**), a certain number of bits are dedicated to red, a certain number to green, and a certain number to blue. Figure 11.4 represents a hicolor pixel.

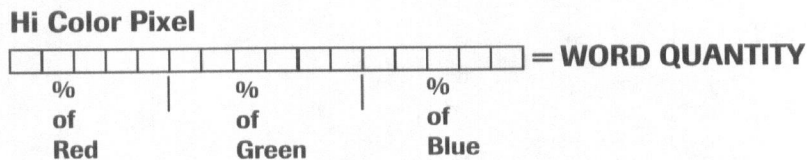


Figure 11.4 A hicolor pixel.

Figure 11.4 uses a bit proportion of 5-6-5 to represent the color in each pixel. If every video card used this bit proportion to represent RGB, the programmer's

task would be simplified. Unfortunately, video card manufacturers haven't adopted any standards for 16-bit RGB pixels, and the programmer must adjust for various hicolor pixel representations. In Figure 11.4, the proportion of R to G to B might very well be 5-5-5 or 6-5-5, and so on.

THE GETSURFACEDESC FUNCTION

In the example programs in the previous chapters, we have been using the **DDSURFACEDESC** structure to set up drawing surfaces for the **CreateSurface** function and to obtain a surface memory pointer and pitch value through the **Lock** function. Once a surface has been created, we can call the **GetSurfaceDesc** function to return a plethora of information about the drawing surface in a **DDSURFACEDESC** structure. **DDSURFACEDESC** contains a sub structure, **DDPIXELFORMAT**, which we can use to find the correct relationship of RGB within a pixel. Check Appendix A for a full listing of **DDSURFACEDESC**.

```
HRESULT GetSurfaceDesc(
    LPDIRECTDRAWSURFACE lpDDSurface,
    LPDDSURFACEDESC lpDDSurfaceDesc)
```

lpDDSurface points to the **DIRECTDRAWSURFACE** object. **lpDDSurface** points to a **DDSURFACEDESC** structure that will be filled in with surface information. The **DDSURFACEDESC** structure passed to **GetSurfaceDesc** should have the **dwSize** field set to the size of the **DDSURFACEDESC** structure and the **dwFlags** field should contain a validity flag for every field of information that is to be returned. In the following example, **GetSurfaceDesc** will return the height and width of a drawing surface:

```
LPDIRECTDRAWSURFACE lpSurface;
DDSURFACEDESC ddsd;
ddsd.dwSize = sizeof ( ddsd );
ddsd.dwFlags = DDSD_HEIGHT | DDSD_WIDTH;
lpSurface->GetSurfaceDesc ( &ddsd );
//ddsd.dwHeight and ddsd.dwWidth contain the height and width of the
surface
```

A return of **DD_OK** indicates success.

THE DDPIXELFORMAT STRUCTURE

The **DDPIXELFORMAT** structure contains a great deal of information about the pixel format of a given surface. Fields within **DDPIXELFORMAT** can be used to determine the bit depth of a surface, whether the surface is RGB or YUV, and any special capabilities the surface may have, such as alpha-channeling, Z buffering, or pixel format conversion. A full reference to **DDPIXELFORMAT** can be found in Appendix A.

In determining the relative amounts of RGB within a surface pixel, we can use a set of color mask values within **DDPIXELFORMAT**:

```
DWORD dwRBitMask
DWORD dwGBitMask
DWORD dwBBitMask
```

The values within these fields, expressed as logical AND masks, denote the amount of a given color element (5 or 6 bits in hicolor modes) and its position within a pixel quantity. If, for example, the mask fields contain the following values:

```
dwRBitMask = %0111110000000000;
dwGBitMask = %0000001111100000;
dwBBitMask = %0000000000001111;
```

We can determine that the RGB pixel format of the surface is 5-5-5. The **HICOLOR.CPP** program listing provides a full example of how to use the **DDPIXELFORMAT** mask fields.

THE HICOLOR.CPP PROGRAM

```

/*****
//DirectDraw Programming
//Bret Timmins
//
//(C)1995 M&T Books
//
//HICOLOR.cpp
// Displays an image in a hicolor video mode
/*****
#define WIN32_LEAN_AND_MEAN
#include <windows.h>

```



```
#include <windowsx.h>
#if defined( __BORLANDC__ ) && defined( __WIN32__ )
#define _WIN32
#endif
#include <ddraw.h>
#include <stdio.h>
#include <stdlib.h>
#include <stdarg.h>

#define SCREEN_WIDTH 640
#define SCREEN_HEIGHT 480

#define SURFACE_WIDTH 540
#define SURFACE_HEIGHT 272

LPDIRECTDRAW          lpDirectDrawObject; //DD object
LPDIRECTDRAWSURFACE lpPrimary;           //DD primary surface
LPDIRECTDRAWSURFACE lpOffscreen;         //Offscreen surface image
BOOL                  ActiveApp;         //Is this program active?

//RGB16 contains shift values that will be applied to 24 bit
//bmp file
//RGBQUAD Depth - number of bits for a given color element
//(5 or 6)
//RGBQUAD Amount - right shift values
//(proportion of R to G to B)
//8Bit.element>>Amount.element
//RGBQUAD Position - within the word the represents a
//16 bit pixel
typedef struct
{
    RGBQUAD Depth;
    RGBQUAD Amount;
    RGBQUAD Position;
} RGB16;

int PASCAL WinMain( HINSTANCE hInstance, HINSTANCE hPrevInstance,
                    LPSTR lpCmdLine, int nCmdShow);
long FAR PASCAL WindowProc ( HWND hwnd, UINT message,
                             WPARAM wParam, LPARAM lParam );
BOOL GetRGB16 ( LPDIRECTDRAWSURFACE Surface, RGB16 *rgb16 );
BOOL DrawFrame ( void );
BOOL RestoreSurfaces ( void );
BOOL BMPToDirectDrawSurface ( LPDIRECTDRAWSURFACE surface,
                              char *bmp );

//*****
//WinMain - mandatory windows init function
//
//*****
```

```
int PASCAL WinMain( HINSTANCE hInstance, HINSTANCE hPrevInstance,
                   LPSTR lpCmdLine, int nCmdShow)
{
    MSG          msg;
    HWND         hwnd;
    WNDCLASS     wc;
    static char  ClassName[] = "HICOLOR";
    DDSURFACEDESC ddsd;
    DDCOLORKEY   ddck;
    HRESULT      ddreturn;

    lpCmdLine = lpCmdLine;
    hPrevInstance = hPrevInstance;

    //register and realize our display window
    wc.style = CS_HREDRAW | CS_VREDRAW;
    wc.lpfnWndProc = WindowProc;
    wc.cbClsExtra = 0;
    wc.cbWndExtra = 0;
    wc.hInstance = hInstance;
    wc.hIcon = LoadIcon( hInstance, IDI_APPLICATION );
    wc.hCursor = LoadCursor( NULL, IDC_ARROW );
    wc.hbrBackground = NULL;
    wc.lpszMenuName = ClassName;
    wc.lpszClassName = ClassName;
    RegisterClass( &wc );

    //create a full screen window so that GDI won't ever be
    //called
    hwnd = CreateWindowEx(
        WS_EX_TOPMOST,
        ClassName,
        ClassName,
        WS_POPUP,
        0,
        0,
        GetSystemMetrics(SM_CXSCREEN),
        GetSystemMetrics(SM_CYSCREEN),
        NULL,
        NULL,
        hInstance,
        NULL );

    if( !hwnd )
        return FALSE;

    ShowWindow( hwnd, nCmdShow );
    SetFocus( hwnd );
    ShowCursor( FALSE );
}
```

```
//Instantiate our DirectDraw object
ddreturn = DirectDrawCreate( NULL,
                             &lpDirectDrawObject, NULL );
if ( ddreturn != DD_OK )
{
    DestroyWindow( hwnd );
    return FALSE;
}

ddreturn = lpDirectDrawObject->SetCooperativeLevel( hwnd,
                                                    DDSCL_EXCLUSIVE | DDSCL_FULLSCREEN );
if ( ddreturn != DD_OK )
{
    DestroyWindow( hwnd );
    return FALSE;
}

// Set the video mode to 640x480x16
ddreturn = lpDirectDrawObject->SetDisplayMode( SCREEN_WIDTH,
                                                SCREEN_HEIGHT, 16);
if ( ddreturn != DD_OK )
{
    DestroyWindow( hwnd );
    return FALSE;
}

// Create the primary surface
ddsd.dwSize = sizeof( ddsd );
ddsd.dwFlags = DDSCL_CAPS;
ddsd.ddsCaps.dwCaps = DDSCAPS_PRIMARYSURFACE;

ddreturn = lpDirectDrawObject->CreateSurface( &ddsd,
                                              &lpPrimary, NULL );
if ( ddreturn != DD_OK )
{
    DestroyWindow( hwnd );
    return FALSE;
}

// Create the main offscreen surface
ddsd.dwFlags = DDSCL_CAPS | DDSCL_HEIGHT | DDSCL_WIDTH;
ddsd.ddsCaps.dwCaps = DDSCAPS_OFFSCREENPLAIN;
ddsd.dwHeight = SURFACE_HEIGHT;
ddsd.dwWidth = SURFACE_WIDTH;
ddreturn = lpDirectDrawObject->CreateSurface ( &ddsd,
                                              &lpOffscreen,
                                              NULL );
if ( ddreturn != DD_OK )
{
```

```
        DestroyWindow( hwnd );
        return FALSE;
    }

    //Set the transparent color key for the off screen surface
    ddck.dwColorSpaceLowValue = 0;
    ddck.dwColorSpaceHighValue = 0;
    lpOffscreen->SetColorKey( DDCKEY_SRCBLT, &ddck );

    //copy to the offscreen surface
    if ( !BMPToDirectDrawSurface( lpOffscreen, "ddraw.bmp" ) )
    {
        DestroyWindow( hwnd );
        return FALSE;
    }

    DrawFrame ();

    while( 1 )
    {
        if( PeekMessage( &msg, NULL, 0, 0, PM_NOREMOVE ) )
        {
            if( !GetMessage( &msg, NULL, 0, 0 ) )
            {
                return msg.wParam;
            }
            TranslateMessage(&msg);
            DispatchMessage(&msg);
        }
        else if ( !ActiveApp )
        {
            WaitMessage ();
        }
    }
}

//WinMain

//*****
//WindowProc - receive and handle windows messages
//
//*****
long FAR PASCAL WindowProc( HWND hwnd, UINT message,
                           WPARAM wParam, LPARAM lParam )
{
    HDC          hdc;
    PAINTSTRUCT ps;

    switch( message )
    {
        case WM_ACTIVATEAPP:
            ActiveApp = wParam;
            break;
    }
}
```

```

case WM_KEYDOWN:
    switch ( wParam )
    {
        case VK_ESCAPE:
            DestroyWindow ( hwnd );
            break;
    }
    break;

case WM_DESTROY:
    if ( lpDirectDrawObject != NULL )
    {
        if ( lpOffscreen != NULL )
            lpOffscreen->Release();
        if ( lpPrimary != NULL )
            lpPrimary->Release();
        lpDirectDrawObject->Release();
    }

    ShowCursor ( TRUE );
    PostQuitMessage ( wParam );
    break;

case WM_PAINT:
    hdc = BeginPaint ( hwnd, &ps );
    DrawFrame ();
    EndPaint ( hwnd, &ps );
    return 0L;

default:
    return DefWindowProc ( hwnd,message,wParam,lParam);
}

return 0L;

} //WindowProc

//*****
//GetRGB16 - returns proportion of RGB within a surface
//          returns left shift represent RBG positions
//input:
// LPDIRECTDRAWSURFACE lpSurface - ptr to 16 bit surface object
// RGB16 *rgb - structure that will get 16 bit RGB shift info
//returns:
// TRUE - success
//*****
BOOL GetRGB16 ( LPDIRECTDRAWSURFACE Surface, RGB16 *rgb16 )
{
    HRESULT ddreturn;

```

```

DDSURFACEDESC ddsd;
BYTE shiftcount;

//get a surface description
ddsd.dwSize = sizeof( ddsd );
ddsd.dwFlags = DDSF_PIXELFORMAT;
if ( ddsd.dwFlags & DDSF_PIXELFORMAT != DD_OK )
    return FALSE;

//get red
shiftcount = 0;
while ( !(ddsd.ddpfPixelFormat.dwRBitMask & 1) )
{
    ddsd.ddpfPixelFormat.dwRBitMask >>= 1;
    shiftcount++;
}
rgb16->Depth.rgbRed = (BYTE) ddsd.ddpfPixelFormat.dwRBitMask;
rgb16->Position.rgbRed = shiftcount;
rgb16->Amount.rgbRed = (ddsd.ddpfPixelFormat.dwRBitMask
    == 0x1f) ? 3 : 2;

//get green
shiftcount = 0;
while ( !(ddsd.ddpfPixelFormat.dwGBitMask & 1) )
{
    ddsd.ddpfPixelFormat.dwGBitMask >>= 1;
    shiftcount++;
}
rgb16->Depth.rgbGreen = (BYTE) ddsd.ddpfPixelFormat.dwGBitMask;
rgb16->Position.rgbGreen = shiftcount;
rgb16->Amount.rgbGreen = (ddsd.ddpfPixelFormat.dwGBitMask
    == 0x1f) ? 3 : 2;

//get Blue
shiftcount = 0;
while ( !(ddsd.ddpfPixelFormat.dwBBitMask & 1) )
{
    ddsd.ddpfPixelFormat.dwBBitMask >>= 1;
    shiftcount++;
}
rgb16->Depth.rgbBlue = (BYTE) ddsd.ddpfPixelFormat.dwBBitMask;
rgb16->Position.rgbBlue = shiftcount;
rgb16->Amount.rgbBlue = (ddsd.ddpfPixelFormat.dwBBitMask
    == 0x1f) ? 3 : 2;

return TRUE;

} //GetRBG16

/*****

```

```
//DrawFrame - Compose a frame of graphics, flip surfaces
//
//return:
//  BOOL TRUE - success
//*****
BOOL DrawFrame ( void )
{
    HRESULT ddreturn;
    DDBLTFX BltFX;
    static RECT ImageRect = {0,0,SURFACE_WIDTH,SURFACE_HEIGHT};
    RECT AnimRect,ColorRect;
    static int XPos = 50, YPos = 100;
    int Color,i;
    BYTE CRed=0,CGreen=0,CBlue=0;
    RGB16 rgb16;

    if ( !GetRBG16 ( lpPrimary, &rgb16 ) )
        return FALSE;

    for (i=0;i<SCREEN_HEIGHT;i+=6)
    {
        BltFX.dwSize = sizeof (BltFX);
        Color = (CRed<<rgb16.Position.rgbRed)|
                (CGreen<<rgb16.Position.rgbGreen)|
                (CBlue<<rgb16.Position.rgbBlue);
        BltFX.dwFillColor = Color;

        CBlue++;
        if (CBlue == rgb16.Depth.rgbBlue+1)
        {
            CBlue--;
            CGreen++;
            if (CGreen == rgb16.Depth.rgbGreen+1)
            {
                CGreen--;
                CRed++;
                if (CRed == rgb16.Depth.rgbRed+1)
                {
                    CBlue = 0;
                    CGreen = 0;
                    CRed = 0;
                }
            }
        }
    }

    ColorRect.left = 0;
    ColorRect.top = i;
    ColorRect.right = SCREEN_WIDTH;
    ColorRect.bottom = i+6;
    do
```

```

{
    ddreturn = lpPrimary->Blt ( &ColorRect, NULL, NULL,
                               DDBLT_COLORFILL|DDBLT_WAIT,
                               &BltFX);
    if ( ddreturn == DDERR_SURFACELOST )
    {
        if ( !RestoreSurfaces() )
            return FALSE;
    }
}while ( ddreturn != DD_OK );
}

//Blt the offscreen surface to the back buffer
AnimRect.left = XPos;
AnimRect.top = YPos;
AnimRect.right = XPos+SURFACE_WIDTH;
AnimRect.bottom = YPos+SURFACE_HEIGHT;
do
{
    ddreturn = lpPrimary->Blt( &AnimRect,
                              lpOffscreen, &ImageRect,
                              DDBLT_KEYSRC|DDBLT_WAIT, NULL );
    if ( ddreturn == DDERR_SURFACELOST )
    {
        if ( !RestoreSurfaces() )
            return FALSE;
    }
}while ( ddreturn != DD_OK );

return TRUE;

} //DrawFrame

//*****
//RestoreSurfaces - restores all lost surfaces
//
//returns:
//  BOOL True - success
//*****
BOOL RestoreSurfaces ( void )
{
    HRESULT ddreturn;

    ddreturn = lpPrimary->Restore();
    if ( ddreturn != DD_OK )
    {
        return FALSE;
    }

    ddreturn = lpOffscreen->Restore();

```



```

    if( ddreturn == DD_OK )
    {
        if ( !BMPToDirectDrawSurface (lpOffscreen,"ddraw.bmp"))
            return FALSE;
    }else
    {
        return FALSE;
    }

    return TRUE;

} //RestoreSurfaces

//*****
//BMPToDirectDrawSurface - opens and copies a BMP to a DD surface
// Reads 8 or 24 bit bmp images
//input:
// LPDIRECTDRAWSURFACE surface
// char *bmp - filename
//returns:
// BOOL TRUE - success
//*****
BOOL BMPToDirectDrawSurface ( LPDIRECTDRAWSURFACE surface,
                               char *bmp )
{
    HRESULT          ddreturn;
    DWORD            actualRead;
    HANDLE           hfile;
    char             path[MAX_PATH];
    BITMAPFILEHEADER BMPFileHead;
    BITMAPINFOHEADER BMPFileInfo;
    RGBQUAD          Palette[256];
    PALETTEENTRY     pe[256];
    LPDIRECTDRAWPALETTE lpDDPalette;
    DDSURFACEDESC    ddsd;
    int              i,src_width,t;
    BYTE             *lpDDMemory,*lpBMPMemory,*image;
    WORD             pixel;
    int              r,g,b;
    RGB16            rgb16;

    hfile = CreateFile( bmp, GENERIC_READ, FILE_SHARE_READ,
                       (LPSECURITY_ATTRIBUTES)NULL,
                       OPEN_EXISTING, FILE_ATTRIBUTE_NORMAL,
                       (HANDLE) NULL );

    if ( hfile == INVALID_HANDLE_VALUE )
    {
        strcpy (path,"..\\");           //back up one directory
        strcat( path, bmp );
    }

```

```
hfile = CreateFile( path, GENERIC_READ, FILE_SHARE_READ,
                    (LPSECURITY_ATTRIBUTES) NULL,
                    OPEN_EXISTING, FILE_ATTRIBUTE_NORMAL,
                    (HANDLE) NULL );
if ( hfile == INVALID_HANDLE_VALUE )
    return FALSE;
}

// Read the BMP header and info structures
if( !ReadFile ( hfile, &BMPFileHead, sizeof ( BMPFileHead ),
                &actualRead, NULL ) )
    return FALSE;

if( !ReadFile ( hfile, &BMPFileInfo, sizeof ( BMPFileInfo ),
                &actualRead, NULL ) )
    return FALSE;

//Make sure the BMP is either 8 bit or 24 bit color
if ( BMPFileInfo.biBitCount != 8
    && BMPFileInfo.biBitCount != 24 )
    return FALSE;

//If 8 bit image, get the BMP palette
if ( BMPFileInfo.biBitCount == 8 )
{
    if ( !ReadFile ( hfile, Palette, sizeof( Palette ),
                    &actualRead, NULL ) )
        return FALSE;
}

//Allocate memory for image data
image = (BYTE *) LocalAlloc ( LPTR,
                              BMPFileInfo.biWidth *
                              BMPFileInfo.biHeight *
                              ( BMPFileInfo.biBitCount/8 ) );

if( image == NULL )
    return FALSE;

//read BMP into memory
if( !ReadFile ( hfile, image,
                BMPFileInfo.biWidth * BMPFileInfo.biHeight *
                ( BMPFileInfo.biBitCount/8 ),
                &actualRead, NULL ) )
{
    LocalFree( image );
    return FALSE;
}

//get correct shift values for surface
if ( !GetRGB16 ( surface, &rgb16 ) )
```

```

{
    LocalFree( image );
    CloseHandle(hfile);
    return FALSE;
}

//copy BMP data to DirectDraw surface
memset ( &ddsd, 0, sizeof( DDSURFACEDESC ) );
ddsd.dwSize = sizeof( ddsd );
ddreturn = surface->Lock( NULL, &ddsd, 0, NULL );

if( ddreturn != DD_OK )
{
    LocalFree( image );
    return FALSE;
}

lpDDMemory = (BYTE *) ddsd.lpSurface;
lpBMPMemory = image+((BMPFileInfo.biHeight-1)*
                    BMPFileInfo.biWidth *
                    ( BMPFileInfo.biBitCount/8 ));

if ( BMPFileInfo.biWidth > ddsd.lPitch )
    src_width = ddsd.lPitch;
else
    src_width = BMPFileInfo.biWidth;

if ( BMPFileInfo.biBitCount == 8 )
{
    for( i = 0; i < BMPFileInfo.biHeight; i++ )
    {
        memcpy ( lpDDMemory, lpBMPMemory, src_width );
        lpDDMemory += ddsd.lPitch;
        lpBMPMemory -= src_width;
    }
    //create a DirectDraw palette for the surface
    for( i=0; i<256; i++ )
    {
        pe[i].peRed = Palette[i].rgbRed;
        pe[i].peGreen = Palette[i].rgbGreen;
        pe[i].peBlue = Palette[i].rgbBlue;
    }
    ddreturn=lpDirectDrawObject->CreatePalette(DDPCAPS_8BIT,
                                                pe,
                                                &lpDDPalette,
                                                NULL );

    if ( ddreturn != DD_OK )
        return FALSE;

    //Attach palette to surface

```

```

        surface->SetPalette ( lpDDPalette );
    }
    else
    {
        for( i = 0; i < BMPFileInfo.biHeight; i++ )
        {
            for( t = 0; t < src_width; t++ )
            {
                b = lpBMPMemory[t*3+0]>>rgb16.Amount.rgbBlue;
                g = lpBMPMemory[t*3+1]>>rgb16.Amount.rgbGreen;
                r = lpBMPMemory[t*3+2]>>rgb16.Amount.rgbRed;
                pixel = (r<<rgb16.Position.rgbRed)|
                    (g<<rgb16.Position.rgbGreen)|
                    (b<<rgb16.Position.rgbBlue);
                lpDDMemory[t*2+1] = (BYTE) (pixel>>8);
                lpDDMemory[t*2+0] = (BYTE) (pixel & 0xff);
            }
            lpDDMemory += ddsd.lPitch;
            lpBMPMemory -= src_width*3;
        }

        surface->Unlock ( NULL );
        LocalFree( image );
        CloseHandle(hfile);

        return TRUE;

    } //BMPToDirectDrawSurface

```

In the **WinMain** function, we create two surfaces, a primary surface and an off-screen surface, to hold the image that we will blt to the screen. These surfaces require about 908K of video memory. If you are using a 512 K or a 1 MB video card and can't run this program, try changing the **ddscaps** flag for the off-screen surface from:

```
ddsd.ddsCaps.dwCaps = DDSCAPS_OFFSCREENPLAIN;
```

to:

```
ddsd.ddsCaps.dwCaps = DDSCAPS_SYSTEMMEMORY;
```

The **BMPToDirectDrawSurface** function has been modified to accept 8 or 24 bit BMP files. If the image contains 8-bit pixel data, determined by checking the

biBitCount field within the BMPINFOHEADER structure, we copy the palette information from the BMP file to a PALETTEENTRY structure, call **CreatePalette** to establish a DirectDrawPalette object, and then attach the palette to the drawing surface with a call to **SetPalette**.

If the BMP file is a 24-bit RGB image, we call the **GetRGB16** function to seed an internal structure, **RGB16**, with logical shift and position values for converting a 24-bit image to the video card's 16 bit RGB format. If, for example, the **GetRGB16** function returns the following values for red:

```
RGB16.Amount.rgbRed = 3;
RGB16.Position.rgbRed=10;
```

We can use the Amount field to determine how much red information we must remove for the current surface, as in :

```
CurrentRed = BMPRed8Bit >> RGB16.Amount.rgbRed;
```

Once we have reduced the color element, we can use the **RGB16.Position** field to determine where to place the color element within a word pixel:

```
PIXEL = CurrentRed << RGB16.Position.rgbRed;
```

In **BMPToDirectDrawSurface**, the following FOR loop handles the conversion:

```
for( i = 0; i < BMPFileInfo.biHeight; i++ )
{
    for ( t = 0; t < src_width; t++ )
    {
        b = lpBMPMemory[t*3+0]>>rgb16.Amount.rgbBlue;
        g = lpBMPMemory[t*3+1]>>rgb16.Amount.rgbGreen;
        r = lpBMPMemory[t*3+2]>>rgb16.Amount.rgbRed;
        pixel = (r<<rgb16.Position.rgbRed)|
                (g<<rgb16.Position.rgbGreen)|
                (b<<rgb16.Position.rgbBlue);
        lpDDMemory[t*2+1] = (BYTE) (pixel>>8);
        lpDDMemory[t*2+0] = (BYTE) (pixel & 0xff);
    }
    lpDDMemory += ddsd.lPitch;
    lpBMPMemory -= src_width*3;
}
```

The **DrawFrame** function uses the **Depth** field of **RGB16** to build a simple, incremental color ramp of blue intensities. The shades of color are blt'd to the primary surface using a color fill operation (see Chapter 9, "Blting: Blt Effects"). The foreground image is then blitted to the primary surface using a source color key. In palette-independent video modes, the color-key value specifies an actual color intensity value, in this case, zero, or pure black.

Running Windowed

All of the programs in the previous chapters have been designed to run in full-screen, exclusive mode. Full-screen mode is the DirectDraw programmer's method of choice; making page flipping, changeable resolutions and color depths, exclusive palette control more available.

DirectDraw programs can also run in a windowed environment, taking full advantage of Microsoft's Windows 95 user interface and the Win32 GDI, but the added benefits of running DirectDraw in a window don't come without trade-offs. In this chapter, we will write a program that runs the "Windows way" and discuss several different techniques for running windowed.

THE GDI AND THE PRIMARY SURFACE

When we create a primary surface, we aren't actually allocating a new section of video memory. Rather we are given access to a drawing surface that is already in use by the GDI. The primary surface is a GDI shared resource, regardless of the DirectDraw cooperation level. While we cannot "shut down" the GDI (i.e., the GDI's access to the primary surface), we can control its behavior.

Full-screen DirectDraw programs deal with the GDI by creating a window that is sized to the dimensions of the screen and by specifying a window style of `WS_TOPMOST`. Once the GDI instantiates this type of window, the full-screen DirectDraw program effectively controls the GDI, since no other window can appear on screen. The full-screen DirectDraw program can use the GDI functions or it can ignore them altogether, updating the screen with DirectDraw functions or with its own low-level code.

Armed with the knowledge that DirectDraw and the GDI share the primary surface, we can begin to build a DirectDraw program in a window.

RUNNING IN A NORMAL WINDOW

A typical DirectDraw windowed program will use the **DDSCL_NORMAL** **SetCooperativeLevel** setting. The **DDSCL_NORMAL** parameter places several important restrictions upon the DirectDraw program, summarized in the following list:

- *Page Flipping.* Using **DDSCL_NORMAL**, page flippable display surfaces cannot be created. The windowed DirectDraw program writes its graphics directly to the primary surface.
- *Setting Display Modes.* The **DDSCL_NORMAL** program cannot change the current display mode. This means that the DirectDraw program has the same responsibility as any other windows program; it must adjust for various screen resolutions and color depths.
- *Palette Control.* The **DDSCL_EXCLUSIVE** DirectDraw program can alter the entire 256-color palette. The **DDSCL_NORMAL** program is subject to Windows' infamous palette remapping scheme.

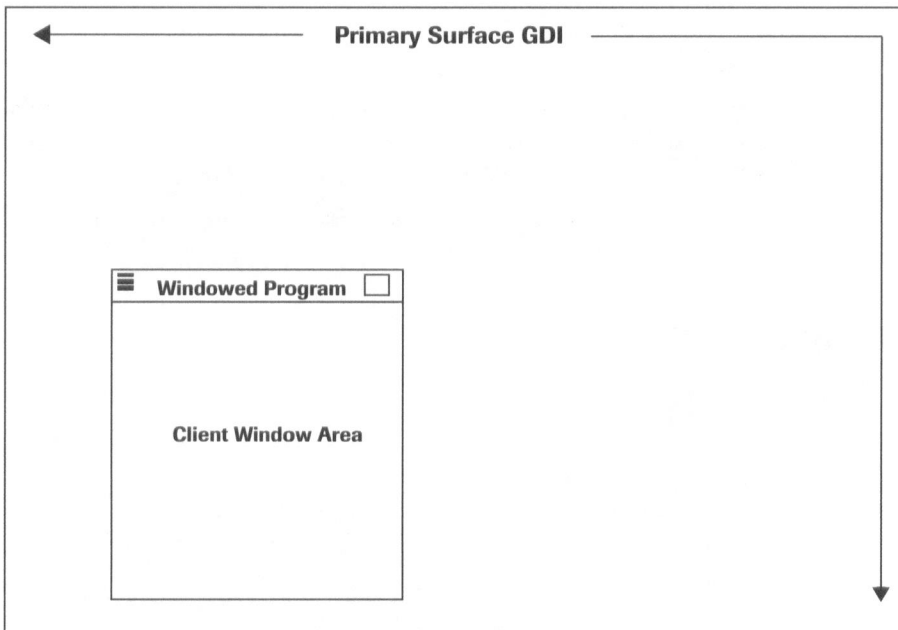
WRITING TO THE PRIMARY SURFACE WINDOW

As stated earlier, a **DDSCL_NORMAL** program cannot create page flippable display surfaces and must write directly to a single primary surface. A windowed DirectDraw program has access to the *entire* primary surface, even though most windowed programs will only want to write data to the section of the primary surface that represents their client window. As an example, consider Figure 12.1.

In Figure 12.1, the DirectDraw program has access to the entire primary surface and could write data anywhere on the screen. This generally undesirable situation can be overcome by attaching a DirectDrawClipper object to the window with the **SetHWND** function. The attached clipper will ensure that the rest of the primary surface is not overwritten by the DirectDraw program. A DirectDraw window with an attached clipper is akin to updating a window with GDI functions.

Windows API functions can also be used to determine the location and size of the DirectDraw program's client window within the primary surface:


```
HWND hwnd;      //The DirectDraw Program's Window
POINT point;
ClientToScreen ( hwnd, &point );
// point.x and point.y hold the starting locations of the Directdraw
program's
//client window area relative to the top of the screen (primary sur-
face)
RECT WindowSize;
GetClientRect ( hwnd, &WindowSize );
//The WindowSize rectangle holds the size of the DirectDraw program's
client window
//relative to the beginning of the window (0,0,width,height). We can
get our final
//window size by adding point.x and point.y to WindowSize
WindowSize.left += point.x;
WindowSize.top += point.y;
WindowSize.right += point.x;
WindowSize.bottom += point.y;
```

**Figure 12.1**

At this point, we have adjusted coordinates representing the client window within the primary surface and could pass these values to the **BltFast** function. Clipping, in this case, rests entirely upon the programmer's shoulders.

USING THE GDI

DirectDraw programs, windowed or full screen, can take full advantage of the GDI functions. A **DirectDrawSurface** object has two member functions, **GetDC**, and **ReleaseDC**, that can be used to create and release device contexts for **DirectDraw** surfaces.

The GetDC Function

```
HRESULT GetDC(
    LPDIRECTDRAWSURFACE lpDirectDrawSurface,
    HDC FAR *lphDC)
```

The first parameter, **lpDirectDrawSurface**, points to the **DirectDrawSurface** object. The second parameter, **lphDC**, points to a handle-to-device context (HDC) that **GetDC** will fill in. A return of **DD_OK** indicates a valid HDC.

The **GetDC** function actually performs a **Lock** on the drawing surface and should be released as soon as possible with a call to **ReleaseDC**.

The ReleaseDC Function

```
HRESULT ReleaseDC(
    LPDIRECTDRAWSURFACE lpDDSsurface,
    HDC hDC)
```

lpDDSsurface points to the **DirectDrawSurface** object. **hDC** is the device context for the surface that was obtained by calling **GetDC**. A return of **DD_OK** indicates a successful release.

FINDING THE GDI

Through the **GetDC/ReleaseDC** mechanism, the GDI can be used to write to any **DirectDraw** surface. In these cases, we are explicitly directing the GDI to use a

DirectDraw surface as the source or destination of its graphical output. But many GDI functions derive their own device contexts from a `HWND` input parameter, and these functions, by implication, write directly to the window represented by the `HWND` variable on the *GDI's primary surface*.

While this doesn't represent a problem for DirectDraw programs running windowed with the **DDSCL_NORMAL** cooperative setting (the primary surface and the visible surface are always the same), it does induce an added wrinkle for full-screen programs that are using page flippable display surfaces. The question becomes one of whether the GDI is writing data to the currently visible display surface.

THE FLIPTOGDISURFACE FUNCTION

While we can't change the surface that the GDI considers the "screen," we can make the GDI's surface the visible surface with a call to the **FlipToGDISurface** function.

```
HRESULT FlipToGDISurface( LPDIRECTDRAW lpDD )
```

`lpDD` points to the DirectDraw object that was returned by the **DirectDrawCreate** function. A return of **DD_OK** indicates that the GDI's primary surface is now the visible surface.

As an example, the following code fragment will ensure that the user will see a message box from a program, regardless of the currently visible display surface:

```
HWND hwnd;           //The program's window
LPDIRECTDRAW lpDD;    //The program's DirectDrawObject
FlipToGDISurface ( lpDD );
MessageBox( hwnd, "Some Message", "A Title", MB_OK );
//The program continues flipping display surfaces...
```

PALETTE CONTROL

Palette manipulation under Windows can be a tricky subject. The Windows Palette Manager (an internal component of the GDI) attempts to moderate palette conflicts between Windows programs by creating a level of abstraction between the Windows program and the actual VGA palette hardware—the logical palette

and the system palette. A logical palette, maintained by the Windows program itself, represents all the color choices that the Windows program would like to have placed into the system palette. If a logical palette represents the ideal palette for a given Windows application, the system palette represents reality.

When a Windows program tries to realize its logical palette (i.e., have its logical palette copied to the system palette), the Windows Palette Manager will attempt to remap the new palette data to fit within the existing system palette. System palette remapping occurs, for the most part, when an application tries to change the STATIC color entries (palette indexes 0 through 9 and 246 through 255 are used by Windows for desktop foregrounds and backgrounds, etc.), or when the Palette Manager detects duplicate color values within the logical palette, in which case, it will “collapse” the duplicate colors into one palette entry, thereby making more of the system palette available.

PALETTE CONTROL AND DIRECTDRAW

Full-screen, exclusive access DirectDraw programs completely circumvent the Windows Palette Manager and set the VGA hardware palette through the **CreatePalette/SetPalette** combination. Programs running with the **DDSCL_NORMAL** parameter, however, cannot bypass the Palette Manager and are subject to the palette remapping described earlier. **DDSCL_NORMAL** programs can either allow palette remapping to take place (using the **CreatePalette/SetPalette** functions), or the **DDSCL_NORMAL** program can ignore DirectDraw’s palette functions altogether and create an “identity palette” through the Palette Manager.



NOTE

The sample programs provided by Microsoft in the GSDK illustrate several palette remapping techniques. The **WINDOW.CPP** listing below provides an example of using an identity palette in a DirectDraw program.

THE WINDOW.CPP PROGRAM

```
//*****
//DirectDraw Programming
//Bret Timmins
//
//(C)1995 M&T Books
```

```
//
//WINDOW.cpp
// Displays an image in a Window, can stretch and scroll.
// Uses an identity palette
//*****
#define WIN32_LEAN_AND_MEAN
#include <windows.h>
#include <windowsx.h>
#if defined( __BORLANDC__ ) && defined( __WIN32__ )
#define _WIN32
#endif
#include <ddraw.h>
#include <stdio.h>
#include <stdlib.h>
#include <stdarg.h>
#include <math.h>
#include "resource.h"

#define BUFFER_WIDTH 640
#define BUFFER_HEIGHT 480

//WorldX and WorldY define start positions within
//the 640 x 480 drawing buffer
int WorldX = 0, WorldY = 0;

LPDIRECTDRAW          lpDirectDrawObject; //DD object
LPDIRECTDRAW_SURFACE lpPrimary;           //DD primary surface
LPDIRECTDRAW_SURFACE lpMainBuffer;        //Fully composed screen
LPDIRECTDRAW_SURFACE lpAnimSet;           //Surface animations
LPDIRECTDRAW_PALETTE lpPalette=NULL;
LPDIRECTDRAW_CLIPPER lpClipper;           //Clipper object

BOOL ActiveApp; //Is this program active?

//Global palette
typedef struct pal
{
    WORD Version;
    WORD NumberOfEntries;
    PALETTEENTRY aEntries[256];
} pal;

pal LogicalPalette =
{
    0x300,
    256
};

//Handle to the global palette
HPALETTE hpalApp;
```

```
#define PI 3.141592654
double sine1,sine2,sine3,cosine1,cosine2,cosine3;
int yaw=0,roll=0,pitch=0;
int PosX,PosY,PosZ;

int PASCAL WinMain( HINSTANCE hInstance, HINSTANCE hPrevInstance,
                    LPSTR lpCmdLine, int nCmdShow);
long FAR PASCAL WindowProc ( HWND hwnd, UINT message,
                             WPARAM wParam, LPARAM lParam );
BOOL DrawFrame ( HWND hwnd, int worldx, int worldy );
void CalcRotation ( void );
void Calculate3D (double worldx,double worldy,double worldz);
BOOL RestoreSurfaces ( void );
BOOL BMPToDirectDrawSurface ( LPDIRECTDRAWSURFACE surface,
                              char *bmp );

BOOL GetBMPPalette ( char *bmp );
void ApplyPalette ( RGBQUAD *palette );
void ClearSystemPalette( void );

//*****
//WinMain - mandatory windows init function
//
//*****
int PASCAL WinMain( HINSTANCE hInstance, HINSTANCE hPrevInstance,
                    LPSTR lpCmdLine, int nCmdShow)
{
    MSG          msg;
    HWND         hwnd;
    WNDCLASS     wc;
    static char  ClassName[] = "Window";
    DDSURFACEDESC ddsd;
    DDCOLORKEY   ddck;
    HRESULT      ddreturn;

    lpCmdLine = lpCmdLine;
    hPrevInstance = hPrevInstance;

    //register and realize our display window
    wc.style = CS_HREDRAW | CS_VREDRAW;
    wc.lpfnWndProc = WindowProc;
    wc.cbClsExtra = 0;
    wc.cbWndExtra = 0;
    wc.hInstance = hInstance;
    wc.hIcon = LoadIcon( hInstance, IDI_APPLICATION );
    wc.hCursor = LoadCursor( NULL, IDC_ARROW );
    wc.hbrBackground = (HBRUSH)(COLOR_WINDOW+1);
    wc.lpszMenuName = MAKEINTRESOURCE(IDR_MENU);
    wc.lpszClassName = ClassName;
    RegisterClass( &wc );
```

```
//create a normal window
hwnd = CreateWindowEx(
    0,
    ClassName,
    ClassName,
    WS_OVERLAPPEDWINDOW|WS_VSCROLL|WS_HSCROLL,
    0,
    0,
    400,
    400,
    NULL,
    NULL,
    hInstance,
    NULL );

if( !hwnd )
    return FALSE;

//Is display mode 8-bit?
HDC Screen;
int bpp;
Screen = CreateIC ("Display",NULL,NULL,NULL);
bpp = GetDeviceCaps (Screen,BITSPIXEL);
DeleteDC (Screen);
if ( bpp != 8 )
{
    MessageBox ( hwnd,
        "This Program requires a 256 color video mode",
        "Window", MB_OK );
    DestroyWindow( hwnd );
    return FALSE;
}

//Instantiate our DirectDraw object
ddreturn = DirectDrawCreate( NULL,
    &lpDirectDrawObject, NULL );
if ( ddreturn != DD_OK )
{
    DestroyWindow( hwnd );
    return FALSE;
}

ddreturn = lpDirectDrawObject->SetCooperativeLevel( hwnd,
    DDSCL_NORMAL );
if ( ddreturn != DD_OK )
{
    DestroyWindow( hwnd );
    return FALSE;
}
```

```
// Create the primary surface
ddsd.dwSize = sizeof( ddsd );
ddsd.dwFlags = DDS_DCAPS;
ddsd.ddsCaps.dwCaps = DDSCAPS_PRIMARYSURFACE;

ddreturn = lpDirectDrawObject->CreateSurface( &ddsd,
                                              &lpPrimary, NULL );

if ( ddreturn != DD_OK )
{
    DestroyWindow( hwnd );
    return FALSE;
}

// Create a clipper
ddreturn = lpDirectDrawObject->CreateClipper ( 0, &lpClipper,
                                              NULL );

if ( ddreturn != DD_OK )
{
    DestroyWindow( hwnd );
    return FALSE;
}

// Get coordinates of our window
ddreturn = lpClipper->SetHwnd ( 0, hwnd );
if ( ddreturn != DD_OK )
{
    DestroyWindow( hwnd );
    return FALSE;
}

// Apply Clipper to the primary surface
ddreturn = lpPrimary->SetClipper( lpClipper );
if ( ddreturn != DD_OK )
{
    DestroyWindow( hwnd );
    return FALSE;
}

//create a surface to compose full screens
ddsd.dwFlags = DDS_DCAPS | DDS_HEIGHT | DDS_WIDTH;
ddsd.ddsCaps.dwCaps = DDSCAPS_OFFSCREENPLAIN;
ddsd.dwHeight = BUFFER_HEIGHT;
ddsd.dwWidth = BUFFER_WIDTH;
ddreturn = lpDirectDrawObject->CreateSurface ( &ddsd,
                                              &lpMainBuffer,
                                              NULL );

if ( ddreturn != DD_OK )
{
    DestroyWindow( hwnd );
}
```



```

        return FALSE;
    }

    // Create the main offscreen surface
    ddsd.dwFlags = DDSD_CAPS | DDSD_HEIGHT | DDSD_WIDTH;
    ddsd.ddsCaps.dwCaps = DDSCAPS_OFFSCREENPLAIN;
    ddsd.dwHeight = 56;
    ddsd.dwWidth = 56;
    ddreturn = lpDirectDrawObject->CreateSurface ( &ddsd,
                                                    &lpAnimSet,
                                                    NULL );

    if ( ddreturn != DD_OK )
    {
        DestroyWindow( hwnd );
        return FALSE;
    }

    //Set the transparent color key for the offscreen surface
    ddck.dwColorSpaceLowValue = 0;
    ddck.dwColorSpaceHighValue = 0;
    ddreturn = lpAnimSet->SetColorKey( DDCKEY_SRCBLT, &ddck );
    if ( ddreturn != DD_OK )
    {
        DestroyWindow( hwnd );
        return FALSE;
    }

    //copy to the offscreen surface
    if ( !BMPTToDirectDrawSurface (lpAnimSet, "atom.bmp" ) )
    {
        DestroyWindow ( hwnd );
        return FALSE;
    }

    if ( !GetBMPPalette ( "atom.bmp" ) )
    {
        DestroyWindow( hwnd );
        return FALSE;
    }

    //ShowWindow will send a WM_PALETTECHANGED to the Window
    //proc handler
    ShowWindow( hwnd, nCmdShow );

    while( 1 )
    {
        if( PeekMessage( &msg, NULL, 0, 0, PM_NOREMOVE ) )
        {
            if( !GetMessage( &msg, NULL, 0, 0 ) )

```

```

        return msg.wParam;
    TranslateMessage(&msg);
    DispatchMessage(&msg);
}
else
{
    DrawFrame ( hwnd, WorldX, WorldY );
}
}

} //WinMain

//*****
//WindowProc - receive and handle windows messages
//
//*****
long FAR PASCAL WindowProc( HWND hwnd, UINT message,
                           WPARAM wParam, LPARAM lParam )
{
    HDC      hdc;
    RECT      WinRect;
    static int ScrollRangeX, ScrollRangeY;

    switch( message )
    {
        case WM_ACTIVATEAPP:
            ActiveApp = wParam;
            break;

        case WM_KEYDOWN:
            switch ( wParam )
            {
                case VK_ESCAPE:
                    DestroyWindow ( hwnd );
                    break;
            }
            break;

        case WM_SIZE:
            if (LOWORD ( lParam ) <= BUFFER_WIDTH )
            {
                ShowScrollBar ( hwnd, SB_HORZ, TRUE );
                ScrollRangeX = BUFFER_WIDTH - LOWORD ( lParam );
                SetScrollRange ( hwnd, SB_HORZ, 0, ScrollRangeX,
                               TRUE );
                if (WorldX + LOWORD ( lParam ) > BUFFER_WIDTH )
                    WorldX = BUFFER_WIDTH - LOWORD ( lParam );
            }
            else
                ShowScrollBar ( hwnd, SB_HORZ, FALSE );

            if ( HIWORD ( lParam ) <= BUFFER_HEIGHT )

```

```

{
    ShowScrollBar ( hwnd, SB_VERT, TRUE );
    ScrollRangeY = BUFFER_HEIGHT - HIWORD ( lParam );
    SetScrollRange ( hwnd, SB_VERT, 0, ScrollRangeY,
                    TRUE );
    if (WorldY + HIWORD ( lParam ) > BUFFER_HEIGHT )
        WorldY = BUFFER_HEIGHT - HIWORD ( lParam );
} else
    ShowScrollBar ( hwnd, SB_VERT, FALSE );

break;

case WM_VSCROLL:
    GetClientRect( hwnd, &WinRect );

    switch ( LOWORD ( wParam ) )
    {
        case SB_LINEUP:
            WorldY--;
            if ( WorldY < 0 )
                WorldY++;
            break;

        case SB_LINEDOWN:
            WorldY++;
            if ( WorldY + WinRect.bottom > BUFFER_HEIGHT )
                WorldY--;
            break;

        case SB_PAGEUP:
            WorldY -= BUFFER_HEIGHT/10;
            if (WorldY < 0 )
                WorldY = 0;
            break;

        case SB_PAGEDOWN:
            WorldY += BUFFER_HEIGHT/10;
            if ( WorldY + WinRect.bottom > BUFFER_HEIGHT )
                WorldY = BUFFER_HEIGHT-WinRect.bottom;
            break;

        case SB_THUMBTRACK:
            WorldY = HIWORD ( wParam );
            break;

        case SB_THUMBPOSITION:
            WorldY = HIWORD ( wParam );
            break;
    }
    if ( WorldY != GetScrollPos ( hwnd, SB_VERT ) )

```

```
        SetScrollPos ( hwnd, SB_VERT, WorldY, TRUE );
    return 0L;

case WM_HSCROLL:
    GetClientRect( hwnd, &WinRect );

    switch ( LOWORD ( wParam ) )
    {
        case SB_LINEUP:
            WorldX--;
            if ( WorldX < 0 )
                WorldX++;
            break;

        case SB_LINEDOWN:
            WorldX++;
            if ( WorldX + WinRect.right > BUFFER_WIDTH )
                WorldX--;
            break;

        case SB_PAGEUP:
            WorldX -= BUFFER_WIDTH/10;
            if ( WorldX < 0 )
                WorldX = 0;
            break;

        case SB_PAGEDOWN:
            WorldX += BUFFER_WIDTH/10;
            if ( WorldX + WinRect.right > BUFFER_WIDTH )
                WorldX = BUFFER_WIDTH - WinRect.right;
            break;

        case SB_THUMBTRACK:
            WorldX = HIWORD ( wParam );
            break;

        case SB_THUMBPOSITION:
            WorldX = HIWORD ( wParam );
            break;
    }
    if ( WorldX != GetScrollPos ( hwnd, SB_HORZ ) )
        SetScrollPos ( hwnd, SB_HORZ, WorldX, TRUE );

    return 0L;

case WM_COMMAND:
    switch ( LOWORD ( wParam ) )
    {
        case ID_FILE_EXIT:
            DestroyWindow ( hwnd );
    }
```

```

        break;
    }
    break;

case WM_DESTROY:
    if ( lpDirectDrawObject != NULL )
    {
        if ( lpAnimSet != NULL )
            lpAnimSet->Release();
        if ( lpMainBuffer != NULL )
            lpMainBuffer->Release();
        if ( lpPrimary != NULL )
            lpPrimary->Release();
        lpDirectDrawObject->Release();
    }
    PostQuitMessage ( wParam );
    break;

case WM_PALETTECHANGED:
    if ( (HWND)wParam == hwnd )
        break;
    // Fall through to WM_QUERYNEWPALETTE
case WM_QUERYNEWPALETTE:
    //Install 1:1 identity palette
    if ( hpalApp )
    {
        hdc = GetDC ( hwnd );
        SelectPalette ( hdc, hpalApp, FALSE );
        RealizePalette ( hdc );
        ReleaseDC ( hwnd, hdc );
        return 1L;
    }
    break;

default:
    return DefWindowProc ( hwnd, message, wParam, lParam );
}

return 0L;

} //WindowProc

//*****
//DrawFrame - Compose a frame of graphics, copy frame
//              to window on primary surface
//return;
// BOOL TRUE - success
//*****
BOOL DrawFrame ( HWND hwnd, int worldx, int worldy )
{

```

```
HRESULT ddreturn;
DDBLTFX BltFX;
RECT BufferRect, AnimRect;
DWORD XPos[10];
DWORD YPos[10];
DWORD ZPos[10];
DWORD i, TempX, TempY, TempZ;

//ColorFill the Background
BltFX.dwSize = sizeof (BltFX);
BltFX.dwFillColor = 0;
do
{
    ddreturn = lpMainBuffer->Blt ( NULL, NULL, NULL,
                                   DDBLT_COLORFILL|DDBLT_WAIT,
                                   &BltFX);
    if ( ddreturn == DDERR_SURFACELOST )
    {
        if ( !RestoreSurfaces() )
            return FALSE;
    }
}while ( ddreturn != DD_OK );

yaw+=4;
if (yaw > 359 )
    yaw = 360 - yaw;
pitch+=2;
if (pitch > 359 )
    pitch = 360 - pitch;
roll++;
if (roll > 359 )
    roll = 360 - roll;

CalcRotation ();

Calculate3D (0,56,0 );
XPos[0] = PosX; YPos[0] = PosY; ZPos[0] = PosZ;
Calculate3D (0,-56,0 );
XPos[1] = PosX; YPos[1] = PosY; ZPos[1] = PosZ;
Calculate3D (56,0,0 );
XPos[2] = PosX; YPos[2] = PosY; ZPos[2] = PosZ;
Calculate3D (-56,0,0 );
XPos[3] = PosX; YPos[3] = PosY; ZPos[3] = PosZ;
Calculate3D (0,0,56 );
XPos[4] = PosX; YPos[4] = PosY; ZPos[4] = PosZ;
Calculate3D (0,0,-56 );
XPos[5] = PosX; YPos[5] = PosY; ZPos[5] = PosZ;
Calculate3D (0,200,0 );
XPos[6] = PosX; YPos[6] = PosY; ZPos[6] = PosZ;
Calculate3D (0,-200,0 );
```

```

XPos[7] = PosX; YPos[7] = PosY; ZPos[7] = PosZ;
Calculate3D (-200,0,0 );
XPos[8] = PosX; YPos[8] = PosY; ZPos[8] = PosZ;
Calculate3D (200,0,0 );
XPos[9] = PosX; YPos[9] = PosY; ZPos[9] = PosZ;

//sort anims based on ZPos
for (i=0;i<9;i++)
{
    if (ZPos[i] > ZPos[i+1] )
    {
        TempX = XPos[i];
        TempY = YPos[i];
        TempZ = ZPos[i];
        XPos[i] = XPos[i+1];
        YPos[i] = YPos[i+1];
        ZPos[i] = ZPos[i+1];
        XPos[i+1] = TempX;
        YPos[i+1] = TempY;
        ZPos[i+1] = TempZ;
    }
}

//Blit new a animation frame from the offscreen surface
AnimRect.left = 0;
AnimRect.top = 0;
AnimRect.right = AnimRect.left+56;
AnimRect.bottom = AnimRect.top+56;

//Blit the offscreen surface to the back buffer
for (i=0;i<10;i++)
{
    do
    {
        ddreturn = lpMainBuffer->BlitFast( XPos[i], YPos[i],
            lpAnimSet, &AnimRect,
            DDBLTFAST_SRCOLORKEY|
            DDBLTFAST_WAIT);
        if ( ddreturn == DDERR_SURFACELOST )
        {
            if ( !RestoreSurfaces() )
                return FALSE;
        }
    }while ( ddreturn != DD_OK );
}

//Blit new frame to window
GetClientRect( hwnd, &AnimRect );

POINT pt;

```

```
pt.x = pt.y = 0;

ClientToScreen( hwnd, &pt );

//case: if the current window size is larger that the drawing
//surface blt to a centered position within window
if ( AnimRect.right >= BUFFER_WIDTH )
{
    pt.x = pt.x + ((AnimRect.right - BUFFER_WIDTH)/2);
    AnimRect.right = BUFFER_WIDTH;
    worldx = 0;
}
if ( AnimRect.bottom >= BUFFER_HEIGHT )
{
    pt.y = pt.y + ((AnimRect.bottom - BUFFER_HEIGHT)/2);
    AnimRect.bottom = BUFFER_HEIGHT;
    worldy = 0;
}

BufferRect.left = pt.x;
BufferRect.top = pt.y;
BufferRect.right = BufferRect.left + AnimRect.right;
BufferRect.bottom = BufferRect.top + AnimRect.bottom;

AnimRect.left += worldx;
AnimRect.top += worldy;
AnimRect.right += worldx;
AnimRect.bottom += worldy;

do
{
    ddreturn = lpPrimary->Blit( &BufferRect,
                               lpMainBuffer, &AnimRect,
                               DDBLT_WAIT, NULL );
    if ( ddreturn == DDERR_SURFACELOST )
    {
        if ( !RestoreSurfaces() )
            return FALSE;
    }
}while ( ddreturn != DD_OK );

return TRUE;

} //DrawFrame

//*****
//CalcRotation - calculate rotation factors based on
//Globals: (yaw, roll, pitch)
//
//seeds: sine1,sine2,sine3 cosinel,cosine2,cosine3
```



```
//*****
void CalcRotation ( void )
{
    sine1 = sin ( (yaw*PI) / 180 );
    sine2 = sin ( (roll*PI) / 180);
    sine3 = sin ( (pitch*PI) / 180);
    cosine1 = cos ( (yaw*PI) / 180);
    cosine2 = cos ( (roll*PI) / 180);
    cosine3 = cos ( (pitch*PI) / 180);
} //CalcRotation

//*****
//Calculate3D - from world x,y,z coordinates, rotate and
//translate 3D coordinates into cartesian x,y
//*****
void Calculate3D (double worldx,double worldy,double worldz)
{
    double  x,y,z,xa,ya,za;

    //absolute x
    worldx = worldx * (-1);
    xa = cosine1 * worldx - sine1 * worldz;
    za = sine1 * worldx + cosine1 * worldz;
    ya = cosine2 * worldy - sine2 * xa;
    x = cosine2 * xa + sine2 * worldy;
    z = cosine3 * za - sine3 * ya;
    y = sine3 * za + cosine3 * ya;

    x = x + 320 - 23;
    y = y + 240 - 23;

    PosX = (int)x;
    PosY = (int)y;
    PosZ = (int)z - 23;
} //Calculate3D

//*****
//RestoreSurfaces - restores all lost surfaces
//
//returns:
//  BOOL True - success
//*****
BOOL RestoreSurfaces ( void )
{
    HRESULT ddreturn;

    ddreturn = lpPrimary->Restore();
    if ( ddreturn != DD_OK )
    {
```

```

        return FALSE;
    }
    ddreturn = lpMainBuffer->Restore();
    if( ddreturn != DD_OK )
    {
        return FALSE;
    }

    ddreturn = lpAnimSet->Restore();
    if( ddreturn == DD_OK )
    {
        if ( !BMPToDirectDrawSurface (lpAnimSet, "atom.bmp" ) )
            return FALSE;
    }else
    {
        return FALSE;
    }

    return TRUE;

} //RestoreSurfaces

//*****
//BMPToDirectDrawSurface - opens and copies a BMP to a DD surface
//input:
// LPDIRECTDRAWSURFACE surface
// char *bmp - filename
//returns:
// BOOL TRUE - success
//*****
BOOL BMPToDirectDrawSurface ( LPDIRECTDRAWSURFACE surface,
                               char *bmp )
{
    HRESULT          ddreturn;
    DWORD            actualRead;
    HANDLE            hfile;
    char              path[MAX_PATH];
    BITMAPFILEHEADER BMPFileHead;
    BITMAPINFOHEADER BMPFileInfo;
    RGBQUAD           Palette[256];
    DDSURFCEDESC      ddsd;
    int               i,src_width;
    BYTE              *lpDDMemory,*lpBMPMemory,*image;

    hfile = CreateFile( bmp, GENERIC_READ, FILE_SHARE_READ,
                       (LPSECURITY_ATTRIBUTES)NULL,
                       OPEN_EXISTING, FILE_ATTRIBUTE_NORMAL,
                       (HANDLE) NULL );

```

```
if ( hfile == INVALID_HANDLE_VALUE )
{
    strcpy (path, "..\\");           //back up one directory
    strcat( path, bmp );
    hfile = CreateFile( path, GENERIC_READ, FILE_SHARE_READ,
                        (LPSECURITY_ATTRIBUTES)NULL,
                        OPEN_EXISTING, FILE_ATTRIBUTE_NORMAL,
                        (HANDLE) NULL );
    if ( hfile == INVALID_HANDLE_VALUE )
        return FALSE;
}

// Read the BMP header and info structures
if( !ReadFile ( hfile, &BMPFileHead, sizeof ( BMPFileHead ),
                &actualRead, NULL) )
    return FALSE;

if( !ReadFile ( hfile, &BMPFileInfo, sizeof ( BMPFileInfo ),
                &actualRead, NULL) )
    return FALSE;

//Make sure the BMP is 8-bit color
if ( BMPFileInfo.biBitCount != 8 )
    return FALSE;

//get the BMP palette
if ( !ReadFile ( hfile, Palette, sizeof( Palette ),
                &actualRead, NULL) )
    return FALSE;

//Allocate memory for image data
image = (BYTE *) LocalAlloc ( LPTR,
                              BMPFileInfo.biWidth *
                              BMPFileInfo.biHeight );

if ( image == NULL )
    return FALSE;

//read BMP into memory
if( !ReadFile ( hfile, image,
                BMPFileInfo.biWidth * BMPFileInfo.biHeight,
                &actualRead, NULL) )
{
    LocalFree( image );
    return FALSE;
}

//copy BMP data to DirectDraw surface
memset ( &ddsd, 0, sizeof( DDSURFACEDESC ) );
```

```

ddsd.dwSize = sizeof( ddsd );
ddreturn = surface->Lock( NULL, &ddsd, 0, NULL );

if( ddreturn != DD_OK )
{
    LocalFree( image );
    return FALSE;
}

lpDDMemory = (BYTE *) ddsd.lpSurface;
lpBMPMemory = image+((BMPFileInfo.biHeight-1)*
                    BMPFileInfo.biWidth);

if ( BMPFileInfo.biWidth > ddsd.lPitch )
    src_width = ddsd.lPitch;
else
    src_width = BMPFileInfo.biWidth;

for( i = 0; i < BMPFileInfo.biHeight; i++ )
{
    memcpy ( lpDDMemory, lpBMPMemory, src_width );
    lpDDMemory += ddsd.lPitch;
    lpBMPMemory -= src_width;
}

surface->Unlock ( NULL );
LocalFree( image );
CloseHandle(hfile);

return TRUE;
} //BMPToDirectDrawSurface

//*****
//GetBMPPalette - gets a palette from a 8-bit BMP file
//input:
// LPDIRECTDRAW lpDirectDrawObject - DirectDraw Object
// char *bmp - filename
//returns:
// TRUE - success
//*****
BOOL GetBMPPalette ( char *bmp )
{
    DWORD          actualRead;
    HANDLE          hfile;
    char            path[MAX_PATH];
    BITMAPFILEHEADER BMPFileHead;
    BITMAPINFOHEADER BMPFileInfo;
    RGBQUAD         Palette[256];

```

```

hfile = CreateFile( bmp, GENERIC_READ, FILE_SHARE_READ,
                    (LPSECURITY_ATTRIBUTES) NULL,
                    OPEN_EXISTING, FILE_ATTRIBUTE_NORMAL,
                    (HANDLE) NULL );

if ( hfile == INVALID_HANDLE_VALUE )
{
    strcpy( path, "..\\");           //back up one directory
    strcat( path, bmp );
    hfile = CreateFile( path, GENERIC_READ, FILE_SHARE_READ,
                        (LPSECURITY_ATTRIBUTES) NULL,
                        OPEN_EXISTING, FILE_ATTRIBUTE_NORMAL,
                        (HANDLE) NULL );
    if ( hfile == INVALID_HANDLE_VALUE )
        return NULL;
}

// Read the BMP header and info structures
if( !ReadFile ( hfile, &BMPPFileHead, sizeof ( BMPPFileHead ),
                &actualRead, NULL ) )
    return FALSE;

if( !ReadFile ( hfile, &BMPPFileInfo, sizeof ( BMPPFileInfo ),
                &actualRead, NULL ) )
    return FALSE;

//Make sure the BMP is 8-bit color
if ( BMPPFileInfo.biBitCount != 8 )
    return FALSE;

//get the BMP palette
if ( !ReadFile ( hfile, Palette, sizeof( Palette ),
                &actualRead, NULL ) )
    return FALSE;

ClearSystemPalette ();
ApplyPalette ( Palette );

return TRUE;
} //GetBMPPalette

/*****
//ClearSystemPalette -
// This routine ensures that our logical identity palette
// won't get collapsed or remapped when we realize it
*****/
void ClearSystemPalette ( void )
{

```

```
// A dummy palette setup
struct
{
    WORD Version;
    WORD NumberOfEntries;
    PALETTEENTRY aEntries[256];
} Palette =
{
    0x300,
    256
};

HPALETTE ScreenPalette = 0;
HDC Screen;
int Counter;

//Reset everything in the system palette to black
for(Counter = 0; Counter < 256; Counter++)
{
    Palette.aEntries[Counter].peRed = 0;
    Palette.aEntries[Counter].peGreen = 0;
    Palette.aEntries[Counter].peBlue = 0;
    Palette.aEntries[Counter].peFlags = PC_NOCOLLAPSE;
}

//Create, select, realize, deselect, and delete the palette
Screen = GetDC(NULL);
ScreenPalette = CreatePalette((LOGPALETTE *)&Palette);

if (ScreenPalette)
{
    ScreenPalette = SelectPalette(Screen,ScreenPalette,FALSE);
    RealizePalette(Screen);
    ScreenPalette = SelectPalette(Screen,ScreenPalette,FALSE);
    DeleteObject(ScreenPalette);
}

ReleaseDC ( NULL, Screen);

} //ClearSystemPalette

//*****
//ApplyPalette -
// Creates a logical palette made up of the STATIC COLORS
// and unique palette colors passed in
// RGBQUAD *palette (indexes 10 - 245 )
// ClearSystemPalette should be called before ApplyPalette
// returns:
// (GLOBAL) hpalApp
//*****
```

```
void ApplyPalette ( RGBQUAD *palette )
{
    HDC Screen;
    int Counter;

    Screen = GetDC ( 0 );
    //We're only concered with system entries 0-9 and 246-255
    GetSystemPaletteEntries ( Screen, 0, 256,
                               LogicalPalette.aEntries );
    ReleaseDC ( 0, Screen );

    for ( Counter = 10; Counter < 246; Counter++ )
    {
        //copy the new color table to indexes 10-245 of the
        //identity palette
        LogicalPalette.aEntries[Counter].peRed =
            palette[Counter].rgbRed;
        LogicalPalette.aEntries[Counter].peGreen =
            palette[Counter].rgbGreen;
        LogicalPalette.aEntries[Counter].peBlue =
            palette[Counter].rgbBlue;
        LogicalPalette.aEntries[Counter].peFlags = PC_NOCOLLAPSE;
    }
    hpalApp = CreatePalette((LOGPALETTE *)&LogicalPalette);
}
//ApplyPalette
```

In **WinMain**, we register and create a normal, non-WS_TOPMOST window. We then check to ensure that the current display mode is a 256-color mode. By using custom routines or through the GDI, we could adjust for different video modes. The Microsoft GSDK sample files use the GDI for this purpose. In this program, if we aren't running in an 8-bit video mode, we simply put up a warning message box and exit.

We then establish our DirectDraw object, set a **DDSCL_NORMAL** cooperation level, and create three DirectDraw surfaces, **lpPrimary**, **lpMainBuffer**, and **lpAnimSet**. The primary surface, which we share with the GDI, is clipped to the client size of the program's window with the **SetHWND** function. The clipper object not only ensures that the program can write graphics only to its client window but also takes care of any overlapping windows that might be on top of the client window.

The program uses the **lpMainBuffer** surface as an interim buffer to hold a complete frame of graphics. **lpMainBuffer**, or a portion of it, depending on the size of the client window, is then blted to the primary surface when a frame of graphics has been fully composed. We could blit the small **lpAnimSet** graphic

directly to the primary surface. This would save memory and might be faster, depending on the size of the primary surface client window. On the other hand, using an interim buffer for a full graphics frame greatly reduces the tearing and shearing that would accompany the direct-blit approach. By knowing the absolute size of the interim buffer, we can avoid any clipping of this surface. As with most programming challenges, choose the method that seems correct to you.

The **DrawFrame** function takes advantage of some very simple 3D projection code (the **CalcRotation** and **Calculate3D** functions) to produce a frame of graphics on **lpMainBuffer**. The full frame, or a portion of it, is then blitted to the primary surface. If the client window is smaller than the size of **lpMainBuffer**, the global variables **WorldX** and **WorldY** are used to determine the starting locations within the surface for the blt operation.

In this program, the **BMPToDirectDrawSurface** function no longer extracts palette information from BMP files. The palette data for the program is acquired in the **GetBMPPalette** function. The purpose of this function is to build an identity palette for the application. An identity palette contains the Windows STATIC colors at palette indexes 0 through 9 and 246 through 255. Palette indexes 10 through 245 are free for the application to use. After the **GETBMPPalette** function reads in the palette data, it calls the **ClearSystemPalette** function to prepare an identity palette and then calls the **ApplyPalette** function to combine the system's STATIC colors with the unique colors from the BMP file. This identity palette, stored in the GLOBAL structure palette, is copied to the system palette when the application receives a **WM_QUERYNEWPALETTE** message in the **WinProc** function.

Debugging DirectDraw Programs

One of the most difficult aspects of DirectDraw programming is debugging. Modern C/C++ compilers include robust, feature-rich GDI-based debuggers that contain almost every conceivable debugging option the programmer might need. The problem is that GDI-based debuggers cannot be used with full-screen, page-flipping DirectDraw programs.

In this chapter, we will discuss another type of Windows debugger, the kernel-level debugger, and we'll add code to a full-screen, page-flippable DirectDraw program that will allow debugging under the GDI.

THE GDI DEBUGGER

The debugging programs included with Microsoft's, Borland's, and Watcom's C/C++ compilers use the GDI to create their graphical displays. These debuggers are dependent upon the normal operation of the GDI and, of course, Windows itself. DirectDraw programs do several things that can render GDI-based debuggers useless:

1. *Surface Locks.* When DirectDraw "locks" a *video memory* surface, either with the **Lock** or the **GetDC** functions, an internal Win32 API function, **Win16Mutex**, is called. **Win16Mutex** halts all multitasking within the system. Under normal operation (i.e., non-debugging), the DirectDraw program will only lock a surface for a brief amount of time. After the program has finished writing data to the surface, it will call the **Unlock** or **ReleaseDC** functions, and system-wide multitasking will resume. The debugger's problem with surface locks appears when we try to "step" over a **Lock** or a **GetDC** function—**Win16Mutex** will be called by

DirectDraw, multitasking will be halted, and our entire development system will be "hosed."

2. *GDI Visibility.* In a page-flipping environment, the GDI's primary surface will be visible only part of the time. This creates an obvious problem when setting breakpoints within the DirectDraw program—will the GDI's primary surface be the visible surface when the breakpoint is reached? With a little preplanning, we can eliminate this problem by inserting the **FlipToGDISurface** function before the breakpoint is reached, but this forces many recompilations of the program and is anathema to the on-demand debugging style that most programmers have come to expect.
3. *WM_TOPMOST Windows.* Full-screen, exclusive-access programs must use the WM_TOPMOST window parameter. When the WM_TOPMOST window is sized to fill the entire screen, the debugger's window will never be seen.

A DEBUGGER COMPROMISE

While there are no easy solutions to these problems outlined, we can design our DirectDraw programs to be (almost) fully debuggable under the GDI and also maintain their full-screen, high-performance aspects when running in a stand-alone, non-debug mode.

This approach requires two pieces of code for every area of the program that will give the debugger problems. The following pseudo-code illustrates this concept:

```
#define DEGUG 1

#ifdef DEBUG
    Create a normal window
#else
    Create a WM_TOPMOST window at full screen
#endif
#ifdef DEBUG
    Create a single primary surface (so we can see the GDI)
#else
    Create multiple page flipping surfaces
#endif
#ifdef DEBUG
    Create off screen surfaces in system memory
#else
    Create off screen surfaces in video memory
```

```
#ifdef DEBUG
    Don't call the flip function
#else
    Call Flip
```

By either defining or not defining the `DEBUG` preprocessor equate, we can compile our programs to be GDI debugger friendly or compile them to run as they will when fully debugged, at full-screen and full-speed. While this type of conditional code substitution will allow us to debug most parts of a DirectDraw program, there are a few caveats: the primary surface will always exist in video memory and can't be stepped-over when locked. An additional, though much less important, side-effect of moving surfaces to system memory will be a decrease in blitting speed.

KERNEL-LEVEL DEBUGGERS

Kernel-level debuggers, used mostly for writing system-level programs, device drivers, etc., bypass the GDI and can be used to debug full-screen, page-flippable DirectDraw programs. The catch is that two computers are required. One computer runs the kernel debugger and the other runs the full-screen DirectDraw program. The computers are connected by network or serial cables.

Soft-ICE, a popular debugger from NU-Mega Technologies, allows full debugging of DirectDraw applications. For additional information on Soft-ICE, contact NU-Mega at:

NU-Mega Technologies, Inc.

PO BOX 7780

Nashua, NH 03060-7780

800-4-NUMEGA

603-889-2383

<http://www.numega.com>

Microsoft's Windbg debugger, included with the Win32 SDK, is capable of remote kernel-level debugging. Consult the debugger's on-line help file for information on setting up a remote debugging session.

THE DEBUG.CPP PROGRAM

The following program modifies the TB_SRC.CPP listing from Chapter 8 to be GDI debugger friendly.



NOTE

When compiled for DEBUG mode, some of the colors in the images will be scrambled.

```
//*****
//DirectDraw Programming
//Bret Timmins
//
//(C)1995 M&T Books
//
//DEBUG.CPP
// A DirectDraw program that can be compiled to run full
// screen with page flipping or can be compiled for a GDI
// based debugger.
//*****
#define DEBUG 1

#define WIN32_LEAN_AND_MEAN
#include <windows.h>
#include <windowsx.h>
#ifdef defined( __BORLANDC__ ) && defined( __WIN32__ )
#define _WIN32
#endif
#include <ddraw.h>
#include <stdio.h>
#include <stdlib.h>
#include <stdarg.h>

#define EVEN 0
#define ODD 1
#define NEG 0xffffffff

LPDIRECTDRAW          lpDirectDrawObject; //DD object
LPDIRECTDRAWSURFACE lpPrimary;            //DD primary surface
LPDIRECTDRAWSURFACE lpBackbuffer;        //DD back buffer surface
LPDIRECTDRAWSURFACE lpOffscreen;         //Offscreen surface
LPDIRECTDRAWSURFACE lpOffBufferEven;     //Surface, buffers
//pixel data that would
//be destroyed by BltFast
//for the primary surface
LPDIRECTDRAWSURFACE lpOffBufferOdd;      //Surface, buffers
```

```

//pixel data that would
//be destroyed by BltFast
//for the back buffer
//Is this program active?
BOOL ActiveApp;

int PASCAL WinMain( HINSTANCE hInstance, HINSTANCE hPrevInstance,
                   LPSTR lpCmdLine, int nCmdShow);
long FAR PASCAL WindowProc ( HWND hwnd, UINT message,
                             WPARAM wParam, LPARAM lParam );
BOOL DrawFrame ( void );
BOOL RestoreSurfaces ( void );
BOOL BMPToDirectDrawSurface ( LPDIRECTDRAWSURFACE surface,
                              char *bmp );

/*****
//WinMain - mandatory windows init function
//
*****/
int PASCAL WinMain( HINSTANCE hInstance, HINSTANCE hPrevInstance,
                   LPSTR lpCmdLine, int nCmdShow)
{
    MSG msg;
    HWND hwnd;
    WNDCLASS wc;
    static char ClassName[] = "SRC_BLT";
    DDSURFACEDESC ddsd;
    DDSCAPS ddscaps;
    DDCOLORKEY ddck;
    HRESULT ddreturn;

    lpCmdLine = lpCmdLine;
    hPrevInstance = hPrevInstance;

    //register and realize our display window
    wc.style = CS_HREDRAW | CS_VREDRAW;
    wc.lpfnWndProc = WindowProc;
    wc.cbClsExtra = 0;
    wc.cbWndExtra = 0;
    wc.hInstance = hInstance;
    wc.hIcon = LoadIcon( hInstance, IDI_APPLICATION );
    wc.hCursor = LoadCursor( NULL, IDC_ARROW );
    wc.hbrBackground = NULL;
    wc.lpszMenuName = ClassName;
    wc.lpszClassName = ClassName;
    RegisterClass( &wc );

    //Create a full screen window so that GDI won't ever be
    //called
#ifdef DEBUG
        hwnd = CreateWindowEx(

```

```
0,
ClassName,
ClassName,
WS_POPUP,
0,
0,
GetSystemMetrics(SM_CXSCREEN),
GetSystemMetrics(SM_CYSCREEN),
NULL,
NULL,
hInstance,
NULL );

#else
    hwnd = CreateWindowEx(
        WS_EX_TOPMOST,
        ClassName,
        ClassName,
        WS_POPUP,
        0,
        0,
        GetSystemMetrics(SM_CXSCREEN),
        GetSystemMetrics(SM_CYSCREEN),
        NULL,
        NULL,
        hInstance,
        NULL );
#endif

if( !hwnd )
    return FALSE;

ShowWindow( hwnd, nCmdShow );
UpdateWindow( hwnd );
SetFocus( hwnd );
#ifdef DEBUG
    ShowCursor( FALSE );
#endif

//Instantiate our DirectDraw object
ddreturn = DirectDrawCreate( NULL,
                             &lpDirectDrawObject, NULL );
if ( ddreturn != DD_OK )
{
    DestroyWindow( hwnd );
    return FALSE;
}

#ifdef DEBUG
    ddreturn = lpDirectDrawObject->SetCooperativeLevel( hwnd,
        DDSCL_NORMAL );
```

```
        if ( ddreturn != DD_OK )
        {
            DestroyWindow( hwnd );
            return FALSE;
        }
    #else
        ddreturn = lpDirectDrawObject->SetCooperativeLevel( hwnd,
                                                            DDSCD_EXCLUSIVE |
                                                            DDSCD_FULLSCREEN );

        if ( ddreturn != DD_OK )
        {
            DestroyWindow( hwnd );
            return FALSE;
        }
        // Set the video mode to 640x480x8
        ddreturn = lpDirectDrawObject->SetDisplayMode( 640,
                                                         480,
                                                         8 );

        if ( ddreturn != DD_OK )
        {
            DestroyWindow( hwnd );
            return FALSE;
        }
    #endif

    // Create the primary surface and one back buffer surface
    #ifdef DEBUG
        ddsd.dwSize = sizeof( ddsd );
        ddsd.dwFlags = DDSD_CAPS;
        ddsd.ddsCaps.dwCaps = DDSCAPS_PRIMARYSURFACE;
    #else
        ddsd.dwSize = sizeof( ddsd );
        ddsd.dwFlags = DDSD_CAPS | DDSD_BACKBUFFERCOUNT;
        ddsd.ddsCaps.dwCaps = DDSCAPS_PRIMARYSURFACE |
                               DDSCAPS_FLIP |
                               DDSCAPS_COMPLEX;

        ddsd.dwBackBufferCount = 1;
    #endif
    ddreturn = lpDirectDrawObject->CreateSurface( &ddsd,
                                                  &lpPrimary, NULL );

    if ( ddreturn != DD_OK )
    {
        DestroyWindow( hwnd );
        return FALSE;
    }

    #ifndef DEBUG
        //get a surface pointer to our back buffer
        ddsd.ddsCaps.dwCaps = DDSCAPS_BACKBUFFER;
        ddreturn = lpPrimary->GetAttachedSurface ( &ddscaps,
```

```
                                &lpBackbuffer);

    if( ddreturn != DD_OK )
    {
        DestroyWindow ( hwnd );
        return FALSE;
    }
#endif

#ifdef DEBUG
    ddsd.ddsCaps.dwCaps = DDSCAPS_OFFSCREENPLAIN |
                        DDSCAPS_SYSTEMMEMORY;
#else
    ddsd.ddsCaps.dwCaps = DDSCAPS_OFFSCREENPLAIN;
#endif

// Create the main offscreen surface
ddsd.dwFlags = DDSD_CAPS | DDSD_HEIGHT | DDSD_WIDTH;
ddsd.dwHeight = 480;
ddsd.dwWidth = 512;
ddreturn = lpDirectDrawObject->CreateSurface ( &ddsd,
                                                &lpOffscreen,
                                                NULL );

if ( ddreturn != DD_OK )
{
    DestroyWindow( hwnd );
    return FALSE;
}

// Create the offscreen surface buffer EVEN
ddsd.dwFlags = DDSD_CAPS | DDSD_HEIGHT | DDSD_WIDTH;
ddsd.dwHeight = 120;
ddsd.dwWidth = 128;
ddreturn = lpDirectDrawObject->CreateSurface ( &ddsd,
                                                &lpOffBufferEven,
                                                NULL );

if ( ddreturn != DD_OK )
{
    DestroyWindow( hwnd );
    return FALSE;
}

// Create the offscreen surface buffer ODD
ddsd.dwFlags = DDSD_CAPS | DDSD_HEIGHT | DDSD_WIDTH;
ddsd.dwHeight = 120;
ddsd.dwWidth = 128;
ddreturn = lpDirectDrawObject->CreateSurface ( &ddsd,
                                                &lpOffBufferOdd,
                                                NULL );

if ( ddreturn != DD_OK )
{
```



```
        DestroyWindow ( hwnd );
        return FALSE;
    }

    //Set the transparent color key for the off screen surface
    ddck.dwColorSpaceLowValue = 255;
    ddck.dwColorSpaceHighValue = 255;
    lpOffscreen->SetColorKey( DDCKEY_SRCBLT, &ddck );

    //copy a BMP file to the primary surface
    if ( !BMPToDirectDrawSurface (lpPrimary, "stars.bmp" ) )
    {
        DestroyWindow ( hwnd );
        return FALSE;
    }

    #ifndef DEBUG
        //copy to the back buffer
        if ( !BMPToDirectDrawSurface(lpBackbuffer,"stars.bmp" ) )
        {
            DestroyWindow ( hwnd );
            return FALSE;
        }
    #endif

    //copy to the offscreen surface
    if ( !BMPToDirectDrawSurface (lpOffscreen, "planet.bmp" ) )
    {
        DestroyWindow ( hwnd );
        return FALSE;
    }

    while( 1 )
    {
        if( PeekMessage( &msg, NULL, 0, 0, PM_NOREMOVE ) )
        {
            if( !GetMessage( &msg, NULL, 0, 0 ) )
                return msg.wParam;
            TranslateMessage(&msg);
            DispatchMessage(&msg);
        }else if ( ActiveApp )
        {
            DrawFrame ();
        }else
        {
            WaitMessage ();
        }
    }

}

} //WinMain
```

```

//*****
//WindowProc - receive and handle windows messages
//
//*****
long FAR PASCAL WindowProc( HWND hwnd, UINT message,
                           WPARAM wParam, LPARAM lParam )
{
    switch( message )
    {
        case WM_ACTIVATEAPP:
            ActiveApp = wParam;
            break;

        case WM_CREATE:
            break;

        case WM_KEYDOWN:
            switch ( wParam )
            {
                case VK_ESCAPE:
                    DestroyWindow ( hwnd );
                    break;
            }
            break;

        case WM_DESTROY:
            if ( lpDirectDrawObject != NULL )
            {
                if ( lpOffBufferEven != NULL )
                    lpOffBufferEven->Release();
                if ( lpOffBufferOdd != NULL )
                    lpOffBufferOdd->Release();
                if ( lpOffscreen != NULL )
                    lpOffscreen->Release();
                if ( lpPrimary != NULL )
                    lpPrimary->Release();
                lpDirectDrawObject->Release();
            }

            #ifndef DEBUG
                ShowCursor( TRUE );
            #endif
            PostQuitMessage ( wParam );
            break;

        default:
            return DefWindowProc (hwnd,message,wParam,lParam);
    }
}
```

```
    return 0L;

} // WindowProc

//*****
// DrawFrame - Compose a frame of graphics, flip surfaces
//
// return:
//  BOOL TRUE - success
//*****
BOOL DrawFrame ( void )
{
    HRESULT ddreturn;
    static RECT rect = { 0, 0, 128, 120 };
    RECT BufferRect, AnimRect;
    static DWORD XPos = 100, YPos = 180;
    static DWORD XOld[2] = { NEG, NEG };
    static DWORD YOld[2] = { NEG, NEG };
    static BOOL Frame = EVEN;
    static LPDIRECTDRAWSURFACE lpBuffer;
    static LONG XStart [16] = { 0,128,256,384,0,128,256,384,
                                0,128,256,384,0,128,256,384 };
    static LONG YStart [16] = { 0,0,0,0,120,120,120,120,
                                240,240,240,240,360,360,360,360 };
    static int AnimFrame = 0;
    static DWORD TickCount = GetTickCount();

    // simple timing
    if ( (GetTickCount() - TickCount) < 50 )
        return TRUE;
    TickCount = GetTickCount();

    if ( Frame == EVEN )
        lpBuffer = lpOffBufferEven;
    else
        lpBuffer = lpOffBufferOdd;

    // Restore back buffer pixels from previous frame
    if ( XOld[Frame] != NEG )
    {
        do
        {
            #ifdef DEBUG
                ddreturn = lpPrimary->BltFast( XOld[Frame],
                                                YOld[Frame],
                                                lpBuffer, &rect,
                                                DDBLTFAST_NOCOLORKEY |
                                                DDBLTFAST_WAIT );
            #else

```

```

        ddreturn = lpBackbuffer->BltFast( XOld[Frame],
                                           YOld[Frame],
                                           lpBuffer, &rect,
                                           DDBLTFAST_NOCOLORKEY|
                                           DDBLTFAST_WAIT );

    #endif
    if ( ddreturn == DDERR_SURFACELOST )
    {
        if ( !RestoreSurfaces() )
            return FALSE;
        //case - after restore, buffers are invalid,
        //rebuild them
        XOld[0] = NEG; XOld[1] = NEG;
        YOld[0] = NEG; YOld[1] = NEG;
        break;
    }
}while ( ddreturn != DD_OK );
}

//Save back buffer pixels to the offscreen buffer
BufferRect.left = XPos;
BufferRect.top = YPos;
BufferRect.right = XPos+128;
BufferRect.bottom = YPos+120;
do
{
    #ifdef DEBUG
        ddreturn = lpBuffer->BltFast( 0, 0,
                                       lpPrimary, &BufferRect,
                                       DDBLTFAST_NOCOLORKEY|DDBLTFAST_WAIT);
    #else
        ddreturn = lpBuffer->BltFast( 0, 0,
                                       lpBackbuffer, &BufferRect,
                                       DDBLTFAST_NOCOLORKEY|DDBLTFAST_WAIT);
    #endif
    if ( ddreturn == DDERR_SURFACELOST )
    {
        if ( !RestoreSurfaces() )
            return FALSE;
    }
}while ( ddreturn != DD_OK );

//Blt new a animation frame from the off screen surface
AnimRect.left = XStart[AnimFrame];
AnimRect.top = YStart[AnimFrame];
AnimRect.right = AnimRect.left+128;
AnimRect.bottom = AnimRect.top+120;

AnimFrame++;
AnimFrame = AnimFrame & 15;

```

```
//Blit the offscreen surface to the back buffer
do
{
    #ifdef DEBUG
    ddreturn = lpPrimary->BlitFast( XPos, YPos,
                                   lpOffscreen, &AnimRect,
                                   DDBLTFAST_SRCCOLORKEY|DDBLTFAST_WAIT );
    #else
    ddreturn = lpBackbuffer->BlitFast( XPos, YPos,
                                       lpOffscreen, &AnimRect,
                                       DDBLTFAST_SRCCOLORKEY|DDBLTFAST_WAIT );
    #endif
    if ( ddreturn == DDERR_SURFACELOST )
    {
        if ( !RestoreSurfaces() )
            return FALSE;
    }
}while ( ddreturn != DD_OK );

//Save old positions for restoring on the next frame
XOld[Frame] = XPos;
YOld[Frame] = YPos;

//set of appropriate buffers for the next frame
if ( Frame == EVEN )
    Frame = ODD;
else
    Frame = EVEN;

#ifdef DEBUG
//Let DirectDraw switch our surface pointers
do
{
    ddreturn = lpPrimary->Flip ( NULL, DDFLIP_WAIT );
    if ( ddreturn == DDERR_SURFACELOST )
    {
        if ( !RestoreSurfaces() )
            return FALSE;
    }
}while ( ddreturn != DD_OK );
#endif

return TRUE;

} //DrawFrame

/*****
//RestoreSurfaces - restores all lost surfaces
//
//returns:
```

```
// BOOL True - success
//*****
BOOL RestoreSurfaces ( void )
{
    HRESULT ddreturn;

    ddreturn = lpPrimary->Restore();
    if ( ddreturn == DD_OK )
    {
        if ( !BMPToDirectDrawSurface (lpPrimary, "stars.bmp") )
            return FALSE;
    }
    }else
    {
        return FALSE;
    }
    }
    #ifndef DEBUG
        ddreturn = lpBackbuffer->Restore();
        if( ddreturn == DD_OK )
        {
            if ( !BMPToDirectDrawSurface(lpBackbuffer,
                                         "stars.bmp") )
                return FALSE;
        }
        }else
        {
            return FALSE;
        }
    }
    #endif
    ddreturn = lpOffscreen->Restore();
    if( ddreturn == DD_OK )
    {
        if ( !BMPToDirectDrawSurface (lpOffscreen,
                                         "planet.bmp") )
            return FALSE;
    }
    }else
    {
        return FALSE;
    }
    }

    ddreturn = lpOffBufferEven->Restore();
    if( ddreturn != DD_OK )
        return FALSE;

    ddreturn = lpOffBufferOdd->Restore();
    if( ddreturn != DD_OK )
        return FALSE;

    return TRUE;
} //RestoreSurfaces
```

```
//*****
//BMPToDirectDrawSurface - opens and copies a BMP to a DD surface
//input:
// LPDIRECTDRAWSURFACE surface
// char *bmp - filename
//returns:
// BOOL TRUE - success
//*****
BOOL BMPToDirectDrawSurface ( LPDIRECTDRAWSURFACE surface,
                               char *bmp )
{
    HRESULT          ddreturn;
    DWORD            actualRead;
    HANDLE            hfile;
    char              path[MAX_PATH];
    BITMAPFILEHEADER BMPFileHead;
    BITMAPINFOHEADER BMPFileInfo;
    RGBQUAD           Palette[256];
    PALETTEENTRY      pe[256];
    LPDIRECTDRAWPALETTE lpDDPalette;
    DDSURFACEDESC     ddsd;
    int               i,src_width;
    BYTE              *lpDDMemory,*lpBMPMemory,*image;

    hfile = CreateFile( bmp, GENERIC_READ, FILE_SHARE_READ,
                       (LPSECURITY_ATTRIBUTES)NULL,
                       OPEN_EXISTING, FILE_ATTRIBUTE_NORMAL,
                       (HANDLE) NULL );

    if ( hfile == INVALID_HANDLE_VALUE )
    {
        strcpy (path,"..\\");           //back up one directory
        strcat( path, bmp );
        hfile = CreateFile( path, GENERIC_READ, FILE_SHARE_READ,
                           (LPSECURITY_ATTRIBUTES)NULL,
                           OPEN_EXISTING, FILE_ATTRIBUTE_NORMAL,
                           (HANDLE) NULL );
        if ( hfile == INVALID_HANDLE_VALUE )
            return FALSE;
    }

    // Read the BMP header and info structures
    if( !ReadFile ( hfile, &BMPFileHead, sizeof ( BMPFileHead ),
                   &actualRead, NULL) )
        return FALSE;

    if( !ReadFile ( hfile, &BMPFileInfo, sizeof ( BMPFileInfo ),
                   &actualRead, NULL) )
        return FALSE;
}
```

```
//Make sure the BMP is 8-bit color
if ( BMPFileInfo.biBitCount != 8 )
    return FALSE;

//get the BMP palette
if ( !ReadFile ( hfile, Palette, sizeof( Palette ),
                &actualRead, NULL ) )

    return FALSE;

//Allocate memory for image data
image = (BYTE *) LocalAlloc ( LPTR,
                             BMPFileInfo.biWidth *
                             BMPFileInfo.biHeight );

if( image == NULL )
    return FALSE;

//read BMP into memory
if( !ReadFile ( hfile, image,
               BMPFileInfo.biWidth * BMPFileInfo.biHeight,
               &actualRead, NULL ) )
{
    LocalFree( image );
    return FALSE;
}

//copy BMP data to DirectDraw surface
memset ( &ddsd, 0, sizeof( DDSURFACEDESC ) );
ddsd.dwSize = sizeof( ddsd );
ddreturn = surface->Lock( NULL, &ddsd, 0, NULL );

if( ddreturn != DD_OK )
{
    LocalFree( image );
    return FALSE;
}

lpDDMemory = (BYTE *) ddsd.lpSurface;
lpBMPMemory = image+((BMPFileInfo.biHeight-1)*
                    BMPFileInfo.biWidth);

if ( BMPFileInfo.biWidth > ddsd.lPitch )
    src_width = ddsd.lPitch;
else
    src_width = BMPFileInfo.biWidth;

for( i = 0; i < BMPFileInfo.biHeight; i++ )
{
    memcpy ( lpDDMemory, lpBMPMemory, src_width );
    lpDDMemory += ddsd.lPitch;
}
```



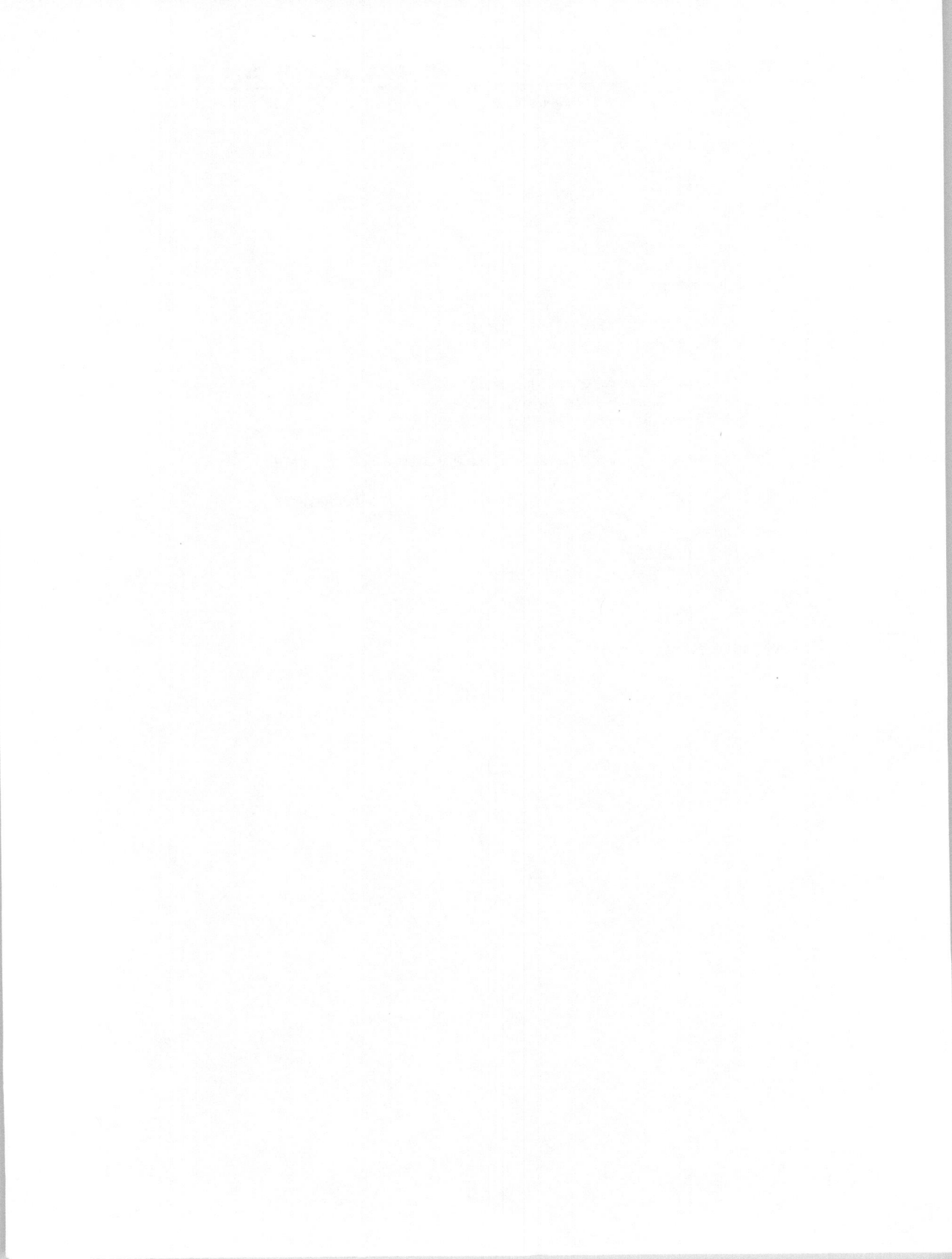
```
        lpBMPMemory -= src_width;
    }

    surface->Unlock ( NULL );
    LocalFree( image );
    CloseHandle(hfile);

    //create a DirectDraw palette for the surface
    for( i=0; i<256; i++ )
    {
        pe[i].peRed = Palette[i].rgbRed;
        pe[i].peGreen = Palette[i].rgbGreen;
        pe[i].peBlue = Palette[i].rgbBlue;
    }
    ddreturn = lpDirectDrawObject->CreatePalette( DDPCAPS_8BIT,
                                                pe,
                                                &lpDDPalette,
                                                NULL );

    if ( ddreturn != DD_OK )
        return FALSE;
    surface->SetPalette ( lpDDPalette );

    return TRUE;
} //BMPToDirectDrawSurface
```



DirectDraw Reference

DIRECTDRAW APIs

DirectDrawCreate

Creates an instance of a DirectDraw object:

```
HRESULT DirectDrawCreate(  
    GUID FAR * lpGUID,  
    LPDIRECTDRAW FAR *lplpDD,  
    IUnknown FAR *pUnkOuter )
```

Parameters:

lpGUID

Points to the GUID representing the driver that should be created. A value of NULL represents the current GUID display driver.

lplpDD

Points to a pointer that will receive the address of the DirectDraw object.

pUnkOuter

Part of the COM interface standard. It should be set to NULL for normal operation.

Return Values:

```
DD_OK  
DDERR_INVALIDPARAMS  
DDERR_INVALIDDIRECTDRAWGUID  
DDERR_GENERIC
```

DDERR_OUTOFMEMORY
DDERR_NODIRECTDRAWHW
DDERR_DIRECTDRAWALREADYCREATED

DirectDrawEnumerate

Through a callback mechanism, returns the number of DirectDraw objects currently running:

```
HRESULT DirectDrawEnumerate(  
    LPDDENUMCALLBACK lpCallback,  
    LPVOID lpContext )
```

Parameters:

lpCallback

Points to a callback function that will be called for each DirectDraw object currently installed. The callback function is prototyped in the following manner:

```
lpCallback(GUID FAR * lpGUID,  
    LPSTR lpDriverDescription,  
    LPSTR lpDriverName,  
    LPVOID lpContext)
```

lpGUID points to a unique identifier for each DirectDraw object. **lpDriverDescription** and **lpDriverName** point to a string representing the DirectDraw object. **lpContext** is a caller-defined pointer.

lpContext

A pointer defined by the calling program. **lpContext** will be passed to the callback function.

Return Values:

DD_OK
DDERR_INVALIDPARAMS

DIRECTDRAW MEMBER FUNCTION REFERENCE

AddRef

Part of the IUnknown interface, **AddRef** is called when a new DirectDraw object is instantiated by an application:

```
DWORD AddRef( LPDIRECTDRAW lpDD )
```

Parameters:

lpDD

Points to the DirectDraw object.

Return Values:

The reference count of the object or zero.

Compact

Not implemented in this version of DirectDraw, the purpose of **Compact** will be to attempt to defragment drawing surface memory (i.e., reallocate surface memory in a continuous segment of memory, if possible). Programs must have obtained exclusive DirectDraw access to use this function:

```
HRESULT Compact(LPDIRECTDRAW lpDD)
```

Parameters:

lpDD

Points to the DirectDraw object.

Return Values:

DD_OK

DDERR_INVALIDOBJECT
DDERR_INVALIDPARAMS
DDERR_SURFACEBUSY
DDERR_NOEXCLUSIVEMODE

CreateClipper

Creates a DirectDrawClipper object:

```
HRESULT CreateClipper(  
    LPDIRECTDRAW lpDD,  
    DWORD dwFlags,  
    LPDIRECTDRAWCLIPPER FAR *lpDDClipper,  
    IUnknown FAR *pUnkOuter )
```

Parameters:

lpDD

Points to the DirectDraw object.

dwFlags

Reserved for future expansion. Should be set to NULL.

lpDDClipper

Points to a pointer that will receive the address of the DirectDrawClipper object.

pUnkOuter

Part of the COM interface and should be set to NULL.

Return Values:

DD_OK
DDERR_INVALIDOBJECT
DDERR_INVALIDPARAMS
DDERR_OUTOFMEMORY
DDERR_NOCOOPERATIVELEVELSET

CREATEPALETTE

Creates a DirectDrawPalette object:

```
HRESULT CreatePalette(  
    LPDIRECTDRAW lpDD,  
    DWORD dwFlags,  
    LPPALETTEENTRY lpColorTable,  
    LPDIRECTDRAWPALETTE FAR* lpDDPalette,  
    IUnknown FAR *pUnkOuter)
```

Parameters:

lpDD

Points to the DirectDraw object.

dwFlags

Indicates the size and type of palette to create:

DDPCAPS_4BIT

Creates a 16-color entry palette (BYTE length)

DDPCAPS_8BITENTRIES

Valid only with **DDPCAPS_4BIT**, creates a 16 color palette that is indexed to an existing 8 bit surface palette.

DDPCAPS_8BIT

Create a 256-color palette.

DDPCAPS_ALLOW256

A full 256 colors can be defined.

DDPCAPS_INITIALIZE

Use the color data passed in **lpColorTable** to initialize the palette.

lpColorTable

Points to a 16- or 256-color table in PALETTEENTRY format.

lp1DDPalette

Points to a pointer that will receive the address of the DirectDrawPalette object.

pUnkOuter

Part of the COM interface and should be set to NULL.

Return Values:

DD_OK
DDERR_INVALIDOBJECT
DDERR_INVALIDPARAMS
DDERR_NOCOOPERATIVELEVELSET
DDERR_NOEXCLUSIVEMODE
DDERR_UNSUPPORTED
DDERR_OUTOFMEMORY
DDERR_OUTOFCAPS

CreateSurface

Creates a DirectDrawSurface object:

```
HRESULT CreateSurface(  
    LPDIRECTDRAW lpDD,  
    LPDDSURFDESC lpDDSurfaceDesc,  
    LPDIRECTDRAWSURFACE FAR *lp1pDDSurface,  
    IUnknown FAR *pUnkOuter )
```

Parameters:

lpDD

Points to the DirectDraw object.

1pDDSurfaceDesc

Points to a **DDSURFACEDESC** that is used to describe to desired characteristics of the surface.

1p1pDDSurface

Points to a pointer that will receive the address of the DirectDrawSurface object.

pUnkOuter

Part of the COM interface and should be set to NULL.

Return Values:

DD_OK
DDERR_INVALIDOBJECT
DDERR_INVALIDPARAMS
DDERR_OUTOFVIDEOMEMORY
DDERR_NODIRECTDRAWHW
DDERR_NOCOOPERATIVELEVELSET
DDERR_INVALIDCAPS
DDERR_INVALIDPIXELFORMAT
DDERR_NOALPHAHW
DDERR_NOFLIPHW
DDERR_NOZBUFFERHW
DDERR_NOEXCLUSIVEMODE
DDERR_OUTOFMEMORY
DDERR_PRIMARYSURFACEALREADYEXISTS
DDERR_NOEMULATION
DDERR_INCOMPATIBLEPRIMARY

DuplicateSurface

Creates a copy of a DirectDrawSurface object from a preexisting DirectDrawSurface object. The newly created object will point to the same surface memory as the original. Primary surfaces, 3D surfaces, and dependent surfaces (back buffers, etc.) cannot be duplicated:

```
HRESULT DuplicateSurface(  
LPDIRECTDRAW lpDD,  
LPDIRECTDRAWSURFACE lpDDSurface,  
LPLPDIRECTDRAWSURFACE lp1pDupDDSurface)
```

Parameters:

lpDD

Points to the DirectDraw object.

lpDDSurface

Points to the DirectDrawSurface object to duplicate.

lp1pDupDDSurface

Points to a pointer that will receive the address of new DirectDrawSurface object.

Return Values:

```
DD_OK  
DDERR_INVALIDOBJECT  
DDERR_INVALIDPARAMS  
DDERR_SURFACELOST  
DDERR_OUTOFMEMORY  
DDERR_CANTDUPLICATE
```

EnumDisplayModes

Through a callback mechanism, returns the number of display modes available to the current DirectDraw object:

```
HRESULT EnumDisplayModes(  
LPDIRECTDRAW lpDD,  
DWORD dwFlags,  
LPDDSURFACEDESC lpDDSurfaceDesc,  
LPVOID lpContext,  
LPDDENUMMODESCALLBACK lpEnumCallback )
```

Parameters:

`lpDD`

Points to the DirectDraw object.

`dwFlags`

Reserved for future expansion and should be set to NULL.

`lpDDSurfaceDesc`

Points to a **DDSURFACEDESC** structure to be checked against available modes. If NULL, all modes will be enumerated.

`lpContext`

A pointer, defined by the calling program. **lpContext** will be passed to the callback function.

`lpEnumCallback`

Points to the function the enumeration procedure will call every time a match is found. The callback function is prototyped in the following manner:

```
lpEnumCallback(  
    LPDDMODEDESC lpDDModeDesc,  
    LPVOID lpContext)
```

lpDDModeDesc points to the structure that contains the mode identifier, monitor frequency, and flags DWORD in addition to the included **DDSURFACEDESC** structure for a mode that provides the necessary functionality. This data is read-only. **lpContext** is the caller-defined pointer. The callback function can return **DDENUMRET_OK** or **DDENUMRET_CANCEL** to continue or stop the mode-enumeration process.

Return Values:

DD_OK
DDERR_INVALIDOBJECT
DDERR_INVALIDPARAMS

EnumSurfaces

Enumerates and, if requested, creates the drawing surfaces that match the input criteria:

```
HRESULT EnumSurfaces(  
    LPDIRECTDRAW lpDD,  
    DWORD dwFlags,  
    LPDDSURFDESC lpDDSD,  
    LPVOID lpContext,  
    LPDDENUMSURFACESCALLBACK lpEnumCallback)
```

Parameters:

lpDD

Points to the DirectDraw object.

dwFlags

DDENUMSURFACES_ALL

All of the surfaces that meet the search criterion will be enumerated.

DDENUMSURFACES_MATCH

Enumerates all surfaces that match the surface description.

DDENUMSURFACES_NOMATCH

Enumerates all surfaces that do not match the surface description.

DDENUMSURFACES_CANBECREATED

Attempts to create the surfaces that match the surface description.

DDENUMSURFACES_DoesExist

Enumerates all surfaces that already exist and match the surface description.

lpDDSD

Points to a **DDSURFACEDESC** structure that defines the search criteria.

lpContext

A pointer, defined by the calling program. **lpContext** will be passed to the callback function.

lpEnumCallback

Points to the function the enumeration procedure will call every time a match is found. The callback function is prototyped in the following manner:

```
lpEnumCallback(  
    LPDIRECTDRAW_SURFACE lpDDSurface,  
    LPDDSURFACEDESC lpDDSurfaceDesc,  
    LPVOID lpContext)
```

If existing surfaces are being enumerated (**DDENUMSURFACES_DoesExist**), then **lpDDSurface** will point to the DirectDrawSurface currently being enumerated. If a potential surface is being enumerated (**DDENUMSURFACES_CanBeCreated**), then the value will be NULL. **lpDDSurfaceDesc** points to the **DDSURFACEDESC** structure for the existing or potential surface that most closely matches the requested surface. **lpContext** points to the caller-defined structure that is passed to the member every time it is invoked. The callback function can return **DDENUMRET_OK** or **DDENUMRET_CANCEL** to continue or stop the surface enumeration process.

Return Values:

DD_OK
DDERR_INVALIDOBJECT
DDERR_INVALIDPARAMS

FlipToGDISurface

Makes the GDI's accessible surface the primary surface:

```
HRESULT FlipToGDISurface(  
    LPDIRECTDRAW lpDD)
```

Parameters:

lpDD

Points to the DirectDraw object.

Return Values:

```
DD_OK  
DDERR_INVALIDPARAMS  
DDERR_INVALIDOBJECT  
DDERR_NOTFOUND
```

GetCaps

Returns the hardware and software emulation capabilities of the DirectDraw object:

```
HRESULT GetCaps(  
    LPDIRECTDRAW lpDD,  
    LPDDCAPS lpDDDriverCaps,  
    LPDDCAPS lpDDHELCaps)
```

Parameters:

lpDD

Points to the DirectDraw object.

lpDDDriverCaps

Points to a **DDCAPS** structure that will receive the capabilities of the current video hardware.

lpDDHELCaps

Points to a **DDCAPS** structure that will receive the software emulation capabilities of the DirectDraw object.

Return Values:

DD_OK
DDERR_INVALIDOBJECT
DDERR_INVALIDPARAMS

GetDisplayMode

Returns the current display mode. Actual display mode changes should be done through **SetDisplayMode** and **RestoreDisplayMode**:

```
HRESULT GetDisplayMode(  
    LPDIRECTDRAW lpDD,  
    LPDDSURFDESC lpDDSurfaceDesc)
```

Parameters:

lpDD

Points to the DirectDraw object.

lpDDSurfaceDesc

Points to a **DDSURFDESC** structure that will be filled in with the display mode information.

Return Values:

DD_OK
DDERR_INVALIDOBJECT
DDERR_INVALIDPARAMS

GetFourCCCodes

```
HRESULT GetFourCCCodes(  
    LPDIRECTDRAW lpDD,  
    DWORD FAR *lpNumCodes,  
    DWORD FAR *lpCodes)
```

Parameters:

lpDD

Points to the DirectDraw object.

lpNumCodes

Points to a DWORD that contains the number of entries returned in the **lpCodes** array. If the number returned is smaller than the actual amount of **FourCC** codes, the **lpCodes** array will contain a truncated amount of codes.

lpCodes

Points to an array of DWORDs that will receive the **FourCC** codes supported by the DirectDraw object. If NULL, then **lpNumCodes** will be set to the number of **FourCC** codes supported by the DirectDraw object.

Return values:

```
DD_OK  
DDERR_INVALIDOBJECT  
DDERR_INVALIDPARAMS
```

GetGDISurface

Returns a DirectDraw surface object pointer to the GDI's primary surface:

```
HRESULT GetGDISurface(  
    LPDIRECTDRAW lpDD,  
    LPDIRECTDRAWSURFACE FAR *lpGDI DDSurface)
```


Parameters:

lpDD

Points to the DirectDraw object.

lp1pGDIIDSSurface

Points to the pointer that will get the address of the **DirectDrawSurface** object the GDI considers the primary surface.

Return values:

DD_OK
DDERR_INVALIDOBJECT
DDERR_INVALIDPARAMS
DDERR_NOTFOUND

GetMonitorFrequency

Returns the frequency of the display monitor for the DirectDraw object, as reported by the Windows registry.

```
HRESULT GetMonitorFrequency(  
    LPDIRECTDRAW lpDD,  
    LPDWORD lpdwFrequency)
```

Parameters:

lpDD

Points to the DirectDraw object.

lpdwFrequency

Points to a DWORD that will receive the frequency. Divide this value by 100 for the actual hertz count.

Return values:

DD_OK
DDERR_INVALIDOBJECT
DDERR_INVALIDPARAMS
DDERR_UNSUPPORTED

GetScanLine

Returns the scan line that is currently being painted to the screen:

```
HRESULT GetScanLine(  
    LPDIRECTDRAW lpDD,  
    LPDWORD lpdwScanLine)
```

Parameters:

lpDD

Points to the DirectDraw object.

lpdwScanLine

Points to a DWORD that will receive the current scan line.

Return Values

DD_OK
DDERR_INVALIDOBJECT
DDERR_INVALIDPARAMS
DDERR_UNSUPPORTED
DDERR_VERTICALBLANKINPROGRESS

GetVerticalBlankStatus

Return TRUE if the display is currently in a vertical blank, FALSE otherwise:

```
HRESULT GetVerticalBlankStatus(  
    LPDIRECTDRAW lpDD,  
    LPBOOL lpbIsInVB)
```

Parameters:

lpDD

Points to the DirectDraw object.

lpbIsInVB

Points to a BOOL that will receive the vertical blank status.

Return values:

DD_OK
DDERR_INVALIDOBJECT
DDERR_INVALIDPARAMS

Initialize

Part of the COM interface, **Initialize** is called when a DirectDraw object is created:

```
HRESULT Initialize(  
    LPDIRECTDRAW lpDD,  
    GUID FAR *lpGUID)
```

Parameters:

lpDD

Points to the DirectDraw object.

lpGUID

Points to the GUID used as the interface identifier.

Return values:

DDERR_ALREADYINITIALIZED

QueryInterface

Part of the IUnknown interface, QueryInterface allows a COM object to increase its capabilities while maintaining binary compatibility with existing client programs. For additional information consult the Microsoft Corp. White paper, "The Component Object Model Specification".

```
HRESULT QueryInterface(  
    LPDIRECTDRAW lpDD,  
    REFIID riid,  
    LPVOID FAR* ppvObj)
```

Parameters:

lpDD

Points to the DirectDraw object.

riid

Points to a UUID. (Universally Unique Identifier).

ppvObj

Points to a pointer that will receive the interface pointer.

Return values:

```
DD_OK  
DDERR_INVALIDOBJECT  
DDERR_INVALIDPARAMS
```

Release

Part of the IUnknown interface, **Release** decreases the reference count of the DirectDraw object:

```
DWORD Release (LPDIRECTDRAW lpDD)
```

Parameters:

lpDD

Points to the DirectDraw object.

Return values:

The reference count of the object or zero.

SetCooperativeLevel

Determines the level of control a DirectDraw application will have in altering shared resources (i.e., the display mode, the hardware palette, the characteristics of the primary surface):

```
HRESULT SetCooperativeLevel(  
    LPDIRECTDRAW lpDD,  
    HWND hWnd,  
    DWORD dwFlags)
```

lpDD

Points to the DirectDraw object.

hWnd

The handle of the DirectDraw applications main window.

dwFlags

Specifies the cooperation level. From the following list:

DDSCL_ALLOWMODEX

Allow the ModeX video modes to be set and allows the creation the ModeX drawing surfaces.

DDSCL_ALLOWREBOOT

Allow CTRL_ALT_DEL the key combination while in full-screen, exclusive mode.

DDSCL_EXCLUSIVE

Obtain exclusive level access. Another DirectDraw application will be unable to set a new display mode or create a page-flippable display.

DDSCL_FULLSCREEN

The application will require the full screen. **DDSCL_EXCLUSIVE/DDSCL_FULLSCREEN** allows page-flippable surfaces to be created.

DDSCL_NORMAL

The application will run as a normal Windows program.

DDSCL_NOWINDOWCHANGES

DirectDraw cannot minimize or restore the application's window.

Return values:

DD_OK
DDERR_INVALIDOBJECT
DDERR_INVALIDPARAMS
DDERR_EXCLUSIVEMODEALREADYSET
DDERR_OUTOFMEMORY
DDERR_HWNDAALREADYSET
DDERR_HWNDSUBCLASSED

SetDisplayMode

Changes the current display mode. An application must have received exclusive access through the **SetCooperationLevel** function for **SetDisplayMode** to succeed. If another DirectDraw application is running when the display mode is changed,

its primary surface will report **DDERR_SURFACELOST**. It must recreate its primary surface to adjust to the new display mode:

```
HRESULT SetDisplayMode(  
    LPDIRECTDRAW lpDD,  
    DWORD dwWidth,  
    DWORD dwHeight,  
    DWORD dwBpp)
```

Parameters:

lpDD

Points to the DirectDraw object.

dwWidth

The width of the new display mode.

dwHeight

The height of the new display mode.

dwBpp

Bits-per-pixel of the new display mode.

Return values:

```
DD_OK  
DDERR_INVALIDOBJECT  
DDERR_INVALIDPARAMS  
DDERR_GENERIC  
DDERR_UNSUPPORTED  
DDERR_INVALIDMODE  
DDERR_LOCKEDSURFACES  
DDERR_WASSTILLDRAWING  
DDERR_SURFACEBUSY  
DDERR_NOEXCLUSIVEMODE
```

WaitForVerticalBlank

Waits for the specified vertical blank condition to be TRUE, or triggers an event when the vertical blank interval begins:

```
HRESULT WaitForVerticalBlank(  
    LPDIRECTDRAW lpDD,  
    DWORD dwFlags,  
    HANDLE hEvent)
```

Parameters:

lpDD

Points to the DirectDraw object.

dwFlags:

DDWAITVB_BLOCKBEGIN

Return on vertical blank start.

DDWAITVB_BLOCKBEGINEVENT

Trigger an event when vertical blank interval begins. Not supported in DirectDraw version 1.0.

DDWAITVB_BLOCKEND

Return on vertical blank end.

hEvent

Event handle.

Return values:

```
DD_OK  
DDERR_INVALIDOBJECT  
DDERR_INVALIDPARAMS  
DDERR_WASSTILLDRAWING  
DDERR_UNSUPPORTED
```


DIRECTDRAWCLIPPER MEMBER FUNCTION REFERENCE

AddRef

Part of the **IUnknown** interface, **AddRef** is called when a new DirectDrawClipper object is instantiated by an application:

```
DWORD AddRef(  
    LPDIRECTDRAWCLIPPER lpDDClipper )
```

Parameters:

lpDDClipper

Points to the DirectDrawClipper object.

Return values:

The reference count of the object or zero.

GetClipList

Returns the clip list, or a portion of the clip list, associated with the DirectDraw Clipper object:

```
HRESULT GetClipList(  
    LPDIRECTDRAWCLIPPER lpDDClipper,  
    LPRECT lpRect,  
    LPRGNDATA lpClipList,  
    LPDWORD lpdwSize )
```

Parameters:

lpDDClipper

Points to the DirectDrawClipper object.

lpRect

If not NULL, **lpClipList** will point to a partial clip list that represents the clipping data within lpRect RECT.

lpClipList

Points to a RGNDATA structure that will receive the complete or partial clip list.

lpdwSize

Points to a DWORD that **GetClipList** will fill with the size of the clip list.

Return values:

DD_OK
DDERR_INVALIDOBJECT
DDERR_INVALIDPARAMS
DDERR_INVALIDCLIPLIST
DDERR_NOCLIPLIST
DDERR_REGIONTOOSMALL
DDERR_GENERIC

GetHWnd

Returns the window handle associated with the DirectDrawClipper object:

```
HRESULT GetHWnd(  
    LPDIRECTDRAWCLIPPER lpDDClipper,  
    HWND FAR *lphWnd )
```

Parameters:

lpDDClipper

Points to the DirectDrawClipper object.

lphWnd

Receives the window handle associated with the DirectDrawClipper object.

Return values:

DD_OK
DDERR_INVALIDOBJECT
DDERR_INVALIDPARAMS

Initialize

Part of the COM interface, **Initialize** is called when a DirectDrawClipper object is created:

```
HRESULT Initialize(  
    LPDIRECTDRAWCLIPPER lpDDClipper,  
    LPDIRECTDRAW lpDD,  
    DWORD dwFlags )
```

Parameters:

lpDDClipper

Points to the DirectDrawClipper object.

lpDD

Points to the DirectDraw object.

dwFlags

Reserved for future use.

Return values:

DDERR_ALREADYINITIALIZED

IsClipListChanged

For a DirectDrawClipper object with an associated HWND, **IsClipListChanged** will return a TRUE if the clip list has changed since it was attached with **SetHWND**:

```
HRESULT IsClipListChanged(  
    LPDIRECTDRAWCLIPPER lpDDClipper,  
    BOOL FAR *lpbChanged )
```

Parameters:

lpDDClipper

Points to the DirectDrawClipper object.

lpbChanged

Points to a BOOL that will receive that status of the clip list.

Return values:

```
DD_OK  
DDERR_INVALIDOBJECT  
DDERR_INVALIDPARAMS
```

QueryInterface

Part of the IUnknown interface, QueryInterface allows a COM object to increase its capabilities while maintaining binary compatibility with existing client programs. For additional information consult the Microsoft Corp. White Paper, "The Component Object Model Specification":

```
HRESULT QueryInterface(  
    LPDIRECTDRAWCLIPPER lpDDClipper,  
    REFIID riid,  
    LPVOID FAR *ppvObj )
```

Parameters:

lpDDClipper

Points to the DirectDrawClipper object.

riid

Points to the GUID interface identifier.

ppvObj

Points to a pointer that will receive the interface pointer.

Return values:

DD_OK
DDERR_INVALIDOBJECT
DDERR_INVALIDPARAMS

Release

Part of the IUnknown interface, **Release** decreases the reference count of the DirectDrawClipper object:

```
DWORD Release( LPDIRECTDRAWCLIPPER lpDDClipper )
```

Parameters:

lpDDClipper

Points to the DirectDrawClipper object.

Return values:

The reference count of the object or zero.

SetClipList

Attaches a clip list to a DirectDrawClipper object:

```
HRESULT SetClipList(  
    LPDIRECTDRAWCLIPPER lpDDClipper,  
    LPRGNDATA lpClipList,  
    DWORD dwFlags )
```

Parameters:

`lpDDClipper`

Points to the `DirectDrawClipper` object.

`lpClipList`

Points to a `RGNDATA` structure representing the clip list. If `NULL`, then an existing clip list associated with the `DirectDrawClipper` object will be deleted.

`dwFlags`

Reserved for future use.

Return values:

`DD_OK`
`DDERR_INVALIDOBJECT`
`DDERR_INVALIDPARAMS`
`DDERR_INVALIDCLIPLIST`
`DDERR_OUTOFMEMORY`
`DDERR_CLIPPERISUSINGHWND`

SetHwnd

Attaches a clip list from a window handle to the `DirectDrawClipper` object. The clip list is updated dynamically to reflect any changes to the window:

```
HRESULT SetHwnd(  
    LPDIRECTDRAWCLIPPER lpDDClipper,  
    DWORD dwFlags,  
    HWND hwnd )
```

Parameters:

`lpDDClipper`

Points to the `DirectDrawClipper` object.

dwFlags

Reserved for future use.

hWnd

Handle to the window.

Return values:

DD_OK
DDERR_INVALIDOBJECT
DDERR_INVALIDPARAMS
DDERR_OUTOFMEMORY

DIRECTDRAWPALETTE MEMBER FUNCTION REFERENCE

AddRef

Part of the IUnknown interface, **AddRef** is called when a new DirectDrawClipper object is instantiated by an application:

```
DWORD AddRef(  
    LPDIRECTDRAWPALETTE lpDDPalette)
```

Parameters:

lpDDPalette

Points to the DirectDrawPalette object.

Return values:

The reference count of the object or zero.

GetCaps

Return the palette capabilities of the DirectDrawPalette object:

```
HRESULT GetCaps(  
    LPDIRECTDRAWPALETTE lpDDPalette,  
    LPDWORD lpdwCaps)
```

Parameters:

lpDDPalette

Points to the DirectDrawPalette object.

lpdwCaps

Points to a DWORD that contains a palette capability flag to check the DirectDrawPalette object against:

```
DDPCAPS_4BIT  
DDPCAPS_8BITENTRIES  
DDPCAPS_8BIT  
DDPCAPS_ALLOW256  
DDPCAPS_PRIMARYSURFACE  
DDPCAPS_PRIMARYSURFACELEFT  
DDPCAPS_VSYNC
```

Return values:

DD_OK will be returned if the DirectDrawPalette has the requested capability.

```
DDERR_INVALIDPARAMS  
DDERR_INVALIDOBJECT
```

GetEntries

Returns palette entries from the DirectDrawPalette object:

```
HRESULT GetEntries(  
    LPDIRECTDRAWPALETTE lpDDPalette,  
    DWORD dwFlags,  
    DWORD dwBase,  
    DWORD dwNumEntries,  
    LPPALETTEENTRY lpEntries )
```


Parameters:

lpDDPalette

Points to the DirectDrawPalette object.

dwFlags

Reserved for future use and must be set to NULL.

dwBase

Defines the starting palette entry to be returned.

dwNumEntries

The number of palette entries to return.

lpEntries

Points to a PALETTEENTRY structure that will receive the palette entries.

Return values:

DD_OK
DDERR_INVALIDOBJECT
DDERR_INVALIDPARAMS
DDERR_NOTPALETTIZED

Initialize

Part of the COM interface, **Initialize** is called when a DirectDrawPalette object is created:

```
HRESULT Initialize(  
    LPDIRECTDRAWPALETTE lpDDPalette,  
    LPDIRECTDRAW lpDD,  
    DWORD dwFlags,  
    LPPALETTEENTRY lpDDColorTable)
```

Parameters:

lpDDPalette

Points to the DirectDrawPalette object.

lpDD

Points to the DirectDraw object.

dwFlags

Not used.

lpDDColorTable

Not used.

Return values:

DDERR_ALREADYINITIALIZED

QueryInterface

Part of the IUnknown interface, QueryInterface allows a COM object to increase its capabilities while maintaining binary compatibility with existing client programs. For additional information consult the Microsoft Corp. White Paper, "The Component Object Model Specification":

```
HRESULT QueryInterface(  
    LPDIRECTDRAWPALETTE lpDDPalette  
    REFIID riid,  
    LPVOID FAR* ppvObj)
```

Parameters:

lpDDPalette

Points to the **DIRECTDRAWPALETTE** structure which was returned to the application when the **DirectDrawPalette** was created.

`riid`

Points to the GUID interface identifier.

`ppvObj`

Points to a pointer which will receive the interface pointer if the request is successful.

Return values:

`DD_OK`
`DDERR_INVALIDPARAMS`
`DDERR_INVALIDOBJECT`

Release

Part of the **IUnknown** interface, **Release** decreases the reference count of the **DirectDrawPalette** object:

```
DWORD Release(  
    LPDIRECTDRAWPALETTE lpDDPalette)
```

Parameters:

`lpDDPalette`

Points to the **DirectDrawPalette** object.

Return values:

The reference count of the object or zero.

SetEntries

Copies palette entry information to the **DirectDrawPalette** object:

```
HRESULT SetEntries(  
    LPDIRECTDRAWPALETTE lpDDPalette,  
    DWORD dwFlags,  
    DWORD dwStartingEntry,  
    DWORD dwCount,  
    LPPALETTEENTRY lpEntries)
```

Parameters:

lpDDPalette

Points to the DirectDrawPalette object.

dwFlags

Reserved for future use and must be set to zero.

dwStartingEnter

Defines the starting palette entry to set.

dwCount

The number of palette entries to set.

lpEntries

Points to a PALETTEENTRY structure that will be copied to the DirectDraw Palette object.

Return values:

```
DD_OK  
DDERR_UNSUPPORTED  
DDERR_INVALIDOBJECT  
DDERR_INVALIDPARAMS  
DDERR_NOTPALETTIZED  
DDERR_NOPALETTEATTACHED
```

DIRECTDRAW_SURFACE MEMBER FUNCTION REFERENCE

AddAttachedSurface

Attaches one surface to another. Video surfaces cannot be attached to system memory surfaces and vice versa. Some surface attachments break existing surface attachments. A surface cannot be attached to itself:

```
HRESULT AddAttachedSurface(  
    LPDIRECTDRAWSURFACE lpDDSurface,  
    LPDIRECTDRAWSURFACE lpDDSAttachedSurface)
```

Parameters:

`lpDDSurface`

Points to the DirectDrawSurface object.

`lpDDSAttachedSurface`

Points to the DirectDrawSurface object that will be attached to **lpDDSurface**.

Return values:

```
DD_OK  
DDERR_INVALIDOBJECT  
DDERR_GENERIC  
DDERR_SURFACELOST  
DDERR_INVALIDPARAMS  
DDERR_SURFACEALREADYATTACHED  
DDERR_WASSTILLDRAWING  
DDERR_CANNOTATTACHSURFACE
```

AddOverlayDirtyRect

Builds a list of rectangles that need to be updated the next time **UpdateOverlay Display** is called. Useful for overlay software emulation only:

```
HRESULT AddOverlayDirtyRect(  
    LPDIRECTDRAWSURFACE lpDDSurface,  
    LPRECT lpRect)
```

Parameters:

lpDDSurface

Points to the DirectDrawSurface object.

lpRect

Points to the RECT that need to be updated.

Return Values:

```
DD_OK  
DDERR_INVALIDPARAMS  
DDERR_INVALIDOBJECT  
DDERR_UNSUPPORTED
```

AddRef

Part of the IUnknown interface, **AddRef** is called when a new DirectDrawSurface object is instantiated by an application.

```
DWORD AddRef(  
    LPDIRECTDRAWSURFACE lpDDSurface)
```

Parameters:

lpDDSurface

Points to the DirectDrawSurface object.

Return values:

The reference count of the object or zero.

Blt

Performs pixel bltting from one drawing surface to another. Depending on the capabilities of the hardware, blts can be performed asynchronously using stretching, Z buffering, alpha channeling, and source and destination color keys, among others:

```
HRESULT Blt(  
    LPDIRECTDRAW_SURFACE lpDDSurface,  
    LPRECT lpDestRect,  
    LPDIRECTDRAW_SURFACE lpDDSrcSurface,  
    LPRECT lpSrcRect,  
    DWORD dwFlags,  
    LPDDBLTFX lpDDBltFx).
```

Parameters:

`lpDDSurface`

Points to the `DirectDrawSurface` object representing the destination of the blt operation.

`lpDestRect`

Points to a `RECT` structure that defines the location and size of the blt on the destination surface.

`lpDDSrcSurface`

Points to the `DirectDrawSurface` object representing the source of the blt operation.

`lpSrcRect`

Points to a `RECT` structure that defines the location and size of the blt on the source surface.

`dwFlags`

NULL if not used, otherwise indicates a special type of blt, define by the following flags:

DDBLT_ALPHADEST

Use the destination surface as the alpha-channel for the blt operation.

DDBLT_ALPHADESTCONSTOVERRIDE

Use the **dwConstAlphaDest** field in **DDBLTFX** the alpha channel override for the blt operation.

DDBLT_ALPHADESTNEG

The destination alpha channel surface becomes more transparent as alpha values increase.

DDBLT_ALPHADESTSURFACEOVERRIDE

Use the **lpDDSAAlphaDest** field in **DDBLTFX** the alpha channel override for the blt operation.

DDBLT_ALPHAEDGEBLEND

Use the **dwAlphaEdgeBlend** field in the **DDBLTFX** structure as the alpha channel for the edges of the image that closely match the color key values.

DDBLT_ALPHASRC

Use the source surface as the alpha channel for the blt operation.

DDBLT_ALPHASRCCONSTOVERRIDE

Use the **dwConstAlphaSrc** field in **DDBLTFX** the alpha channel override for the blt operation.

DDBLT_ALPHASRCNEG

The source alpha channel surface becomes more transparent as alpha values increase.

DDBLT_ALPHASRCSURFACEOVERRIDE

Use the **lpDDSAAlphaSrc** field in **DDBLTFX** the alpha channel override for the blt operation.

DDBLT_ASYNC

Perform the blt asynchronously using the hardware FIFO. No FIFO FULL check is performed.

DDBLT_COLORFILL

Perform a color fill on the destination RECT operation the value in the **DDBLTFX.dwFillColor** field as the color fill value.

DDBLT_DDFX

Specify a blt effects operation using the **DDBLTFX.dwDDFX** field.

DDBLT_DDROPS

Specify a non-defined ROPS using the **DDBLTFX.dwDDROPS** field.

DDBLT_KEYDEST

Perform blt operation using the destination color key.

DDBLT_KEYDESTOVERRIDE

Perform blt operation using the **DDBLTFX.dckDestColorkey** field as the destination color key.

DDBLT_KEYSRC

Perform blt operation using the source color key.

DDBLT_KEYSRCOVERRIDE

Perform blt operation using the **DDBLTFX.dckSrcColorkey** field as the source color key.

DDBLT_ROP

Specify a Win32 API defined ROPS using the **DDBLTFX.dwROP** field.

DDBLT_ROTATIONANGLE

Perform the blt operation using **DDBLTFX.dwRotationAngle** field as rotation angle in 1/100th of a degree.

DDBLT_WAIT

Wait until the blt operation can be performed before returning with a **DDERR_WASSTILLDRAWING** error.

DDBLT_ZBUFFER

Perform a Z buffered blt operation using Z buffers attached to the source and destination drawing surfaces. The **DDBLTFX.dwZBufferOpCode** specifies the Z buffer opcode.

DDBLT_ZBUFFERDESTCONSTOVERRIDE

Perform a Z buffered blt operation using the **DDBLTFX.dwZBufferOpCode** and the **DDBLTFX.dwConstDestZ** as the Z buffer opcode and the Z buffer for the destination surface.

DDBLT_ZBUFFERDESTOVERRIDE

Perform a Z buffered blt operation using the **DDBLTFX.dwZBufferOpCode** and the **DDBLTFX.lpDDSDestZBuffer** as the Z buffer opcode and the Z buffer for the destination surface.

DDBLT_ZBUFFERSRCCONSTOVERRIDE

Perform a Z buffered blt operation using the **DDBLTFX.dwZBufferOpCode** and the **DDBLTFX.dwConstSrcZ** as the Z buffer opcode and the Z buffer for the source surface.

DDBLT_ZBUFFERSRCOVERRIDE

Perform a Z buffered blt operation using the **DDBLTFX.dwZBufferOpCode** and the **DDBLTFX.lpDDSSrcZBuffer** as the Z buffer opcode and the Z buffer for the source surface.

lpDDBltFx

Null if not used, otherwise points to a **DDBLTFX** structure.

Return values:

DD_OK
DDERR_GENERIC
DDERR_INVALIDCLIPLIST
DDERR_INVALIDOBJECT
DDERR_INVALIDPARAMS
DDERR_INVALIDRECT
DDERR_INVALIDRECT
DDERR_NOALPHAHW
DDERR_NOBLTHW
DDERR_NOCLIPLIST
DDERR_NODDROPSHW
DDERR_SURFACELOST
DDERR_UNSUPPORTED
DDERR_NOMIRRORHW
DDERR_NORASTEROPHW
DDERR_NOROTATIONHW
DDERR_NOSTRETCHHW
DDERR_SURFACEBUSY
DDERR_NOZBUFFERHW

BltBatch

Performs multiple **Blt** operations from several sources to a single destination surface. Not supported in DirectDraw V1.0:

```
HRESULT BltBatch(  
    LPDIRECTDRAWSURFACE lpDDDestSurface,  
    LPDDBLTBATCH lpDDBltBatch,  
    DWORD dwCount,  
    DWORD dwFlags)
```

Parameters:

lpDDSurface

Points to the DirectDrawSurface object representing the destination of the blt batch operations.

lpDDBltBatch

Points to a **DDBLTBATCH** structure that define the surface surfaces and type of blts for the bltbatch operation.

dwCount

The number of blts to execute.

dwFlags

Reserved for future expansion.

Return values:

```
DD_OK  
DDERR_GENERIC  
DDERR_INVALIDCLIPLIST  
DDERR_INVALIDOBJECT  
DDERR_INVALIDPARAMS  
DDERR_INVALIDRECT  
DDERR_INVALIDRECT  
DDERR_NOALPHAHW  
DDERR_NOBLTHW  
DDERR_NOCLIPLIST  
DDERR_UNSUPPORTED  
DDERR_SURFACELOST  
DDERR_NODDROPSHW
```

DDERR_NOMIRRORHW
DDERR_NORASTEROPHW
DDERR_NOROTATIONHW
DDERR_NOSTRETCHHW
DDERR_SURFACEBUSY
DDERR_NOZBUFFERHW

BltFast

Performs a pixel blt from one surface to another with no special capabilities except transparency. The blt operation is performed asynchronously if the hardware allows:

```
HRESULT BltFast(  
    LPDIRECTDRAW_SURFACE1 lpDDSurface,  
    DWORD dwX,  
    DWORD dwY,  
    LPDIRECTDRAW_SURFACE1 lpDDSrcSurface,  
    LPRECT lpSrcRect,  
    DWORD dwTrans)
```

Parameters:

`lpDDSurface`

Points to the `DirectDrawSurface` object representing the destination of the blt operation.

`dwX`

Starting X location within the destination.

`dwY`

Starting Y location within the destination.

`lpDDSrcSurface`

Points to the `DirectDrawSurface` object representing the source of the blt operation.

lpSrcRect

Points to a RECT structure that defines the location and size of the blt on the source surface.

dwTrans

The type of blt operation:

DDBLTFAST_DESTCOLORKEY

Perform a transparent blt with the destination color key

DDBLTFAST_NOCOLORKEY

Perform a one to one pixel copy.

DDBLTFAST_SRCCOLORKEY

Perform a transparent blt with the source color key.

DDBLTFAST_WAIT

Wait until the blt operation can be performed before returning with a **DDERR_WASSTILLDRAWING** error.

Return values:

DD_OK
DDERR_INVALIDPARAMS
DDERR_SURFACELOST
DDERR_SURFACEBUSY
DDERR_INVALIDOBJECT
DDERR_INVALIDRECT
DDERR_EXCEPTION
DDERR_UNSUPPORTED
DDERR_GENERIC
DDERR_NOBLTHW

DeleteAttachedSurfaces

Remove the attachments from two drawing surfaces. The surfaces aren't physically released, just returned to their unattached state:

```
HRESULT DeleteAttachedSurfaces(  
    LPDIRECTDRAWSURFACE lpDDSurface,  
    DWORD dwFlags,  
    LPDIRECTDRAWSURFACE lpDDSAttachedSurface)
```

Parameters:

`lpDDSurface`

Points to the `DirectDrawSurface` object.

`dwFlags`

Should be set to zero.

`lpDDSAttachedSurface`

Points to the `DirectDrawSurface` object that will be detached from **`lpDDSurface`**. If it is `NULL`, all surface attachments will be removed.

Return values:

```
DD_OK  
DDERR_INVALIDOBJECT  
DDERR_SURFACELOST  
DDERR_SURFACENOTATTACHED  
DDERR_INVALIDPARAMS  
DDERR_CANNOTDETACHSURFACE
```

EnumAttachedSurfaces

Through a callback mechanism, enumerates all of the surfaces attached to a given surface:

```
HRESULT EnumAttachedSurfaces(  
    LPDIRECTDRAWSURFACE lpDDSurface,  
    LPVOID lpContext,  
    LPDDENUMSURFACESCALLBACK lpEnumSurfacesCallback )
```

Parameters:

`lpDDSurface`

Points to the DirectDrawSurface object.

`lpContext`

A pointer, defined by the calling program. **lpContext** will be passed to the callback function.

`lpEnumSurfacesCallback`

Points to a callback function that will be called for each surface that is attached to `lpDDSurface`. The callback function is prototyped in the following manner:

```
lpEnumSurfacesCallback(  
    LPDIRECTDRAWSURFACE lpDDSurface,  
    LPDDSURFACEDESC lpDDSurfaceDesc,  
    LPVOID lpContext)
```

lpDDSurface points to the attached surface. **lpDDSurface** points to a **DDSURFACEDESC** structure that describes the attached surface. **lpContext** is a caller defined pointer. The callback function can return **DDENUMRET_OK** or **DDENUMRET_CANCEL** to continue or stop the mode enumeration process.

Return Values:

```
DD_OK  
DDERR_INVALIDOBJECT  
DDERR_INVALIDPARAMS  
DDERR_SURFACELOST
```


EnumOverlayZOrders

Through a callback mechanism, enumerates all of the overlays attached to a given surface. Search order can be forwards or backwards.

```
HRESULT EnumOverlayZOrders(  
    LPDIRECTDRAWSURFACE lpDDSurface,  
    DWORD dwFlags,  
    LPVOID lpContext,  
    LPDDENUMSURFACESCALLBACK lpfnCallback)
```

Parameters:

lpDDSurface

Points to the DirectDrawSurface object.

dwFlags:

DDENUMOVERLAYZ_BACKTOFRONT

Enumerate overlay surfaces back to front.

DDENUMOVERLAYZ_FRONTTOBACK

Enumerate overlay surface front to back.

lpContext

A pointer, defined by the calling program. **lpContext** will be passed to the callback function.

lpfnCallback

Points to a callback function that will be called for each overlay that is attached to **lpDDSurface**. The callback function is prototyped in the following manner:

```
lpfnCallback(  
    LPDIRECTDRAWSURFACE lpDDSurface,  
    LPVOID lpContext)
```

lpDDSurface points to the overlay surface. **lpContext** is a caller-defined pointer. The callback function can return **DDENUMRET_OK** or **DDENUMRET_CANCEL** to continue or stop the mode enumeration process.

Return values:

DD_OK
DDERR_INVALIDOBJECT
DDERR_INVALIDPARAMS

Flip

Switches the surface memory associated the current primary surface and the next back buffer in the display buffer ring. The former primary surface memory is now associated with the last DirectDrawSurface object in the display buffer ring. This function is always synchronized to happen during the vertical blank interval:

```
HRESULT Flip(  
    LPDIRECTDRAWSURFACE lpDDSurface,  
    LPDIRECTDRAWSURFACE lpDDSurfaceTargetOverride,  
    DWORD dwFlags)
```

Parameters:

lpDDSurface

Points to the DirectDrawSurface object.

lpDDSurfaceTargetOverride

If **NULL**, then the flip operation will follow the logical order of the display buffer ring. Otherwise, it points to a specific surface within the display buffer ring that will be the new primary surface.

dwFlags

NULL or **DDFLIP_WAIT** (ignore a **DDERR_WASSTILLDRAWING** error and continue until the flip is complete.)

Return values:

DD_OK
DDERR_INVALIDPARAMS
DDERR_INVALIDOBJECT
DDERR_SURFACELOST
DDERR_SURFACEBUSY
DDERR_GENERIC
DDERR_WASSTILLDRAWING
DDERR_UNSUPPORTED
DDERR_NOTFLIPPABLE
DDERR_NOFLIPHW

GetAttachedSurface

Returns an attached surface that matches the requested characteristics. Duplicate attached surfaces will return an error:

```
HRESULT GetAttachedSurface(  
    LPDIRECTDRAWSURFACE lpDDSurface,  
    LPDDSCAPS lpDDSCaps,  
    LPLPDIRECTDRAWSURFACE FAR *lp1pDDAttachedSurface)
```

Parameters:

lpDDSurface

Points to the DirectDrawSurface object.

lpDDSCaps

Points to a **DDSCAPS** structure that contains the search criteria for the attached surface.

lp1pDDAttachedSurface

If the requested attached surface is found, **lp1pDDAttachedSurface** will point to a pointer to the DirectDrawSurface object.

Return Values:

DD_OK
DDERR_INVALIDOBJECT
DDERR_SURFACELOST
DDERR_INVALIDPARAMS
DDERR_NOTFOUND

GetBltStatus

Returns the status of the hardware bltter, as it pertains to the DirectDrawSurface object:

```
HRESULT GetBltStatus(  
    LPDIRECTDRAWSURFACE lpDDSurface,  
    DWORD dwFlags)
```

Parameters:

lpDDSurface

Points to the DirectDrawSurface object.

dwFlags:

DDGBS_CANBLT

Returns **DD_OK** if a blt can be completed on this surface.

DDGBS_ISBLTDONE

Returns **DD_OK** if the last blt on this surface has completed.

Return values:

DD_OK
DDERR_INVALIDPARAMS
DDERR_INVALIDOBJECT
DDERR_SURFACELOST
DDERR_SURFACEBUSY
DDERR_UNSUPPORTED
DDERR_WASSTILLDRAWING
DDERR_NOBLTHW

GetCaps

Returns the capabilities of the DirectDrawSurface object:

```
HRESULT GetCaps(  
    LPDIRECTDRAWSURFACE lpDDSurface,  
    LPDDSCAPS lpDDSCaps)
```

Parameters:

lpDDSurface

Points to the DirectDrawSurface object.

lpDDSCaps

Points to a **DDSCAPS** structure that will receive the capabilities of the DirectDrawSurface object.

Return values:

```
DD_OK  
DDERR_INVALIDOBJECT  
DDERR_INVALIDPARAMS
```

GetClipper

Returns the DirectDrawClipper object associated with the DirectDrawSurface object:

```
HRESULT GetClipper(  
    LPDIRECTDRAWSURFACE lpDDSurface,  
    LPDIRECTDRAWCLIPPER FAR * lpDDClipper)
```

Parameters:

lpDDSurface

Points to the DirectDrawSurface object.

lpDDClipper

Points to a pointer to the DirectDrawClipper object.

Return values:

DD_OK
DDERR_INVALIDOBJECT
DDERR_INVALIDPARAMS
DDERR_NOCLIPPERATTACHED

GetColorKey

Returns the color key associated with the DirectDrawSurface object:

```
HRESULT GetColorKey(  
    LPDIRECTDRAWSURFACE lpDDSsurface,  
    DWORD dwFlags,  
    LPDDCOLORKEY lpDDColorKey)
```

Parameters:

lpDDSsurface

Points to the DirectDrawSurface object.

dwFlags

Determines which color key to return:

DDCKEY_COLORSPACE
DDCKEY_DESTBLT
DDCKEY_DESTOVERLAY
DDCKEY_SRCBLT
DDCKEY_SRCOVERLAY

lpDDColorKey

Will receive the address of the specified color key.

Return values:

DD_OK
DDERR_INVALIDOBJECT
DDERR_INVALIDPARAMS
DDERR_SURFACELOST
DDERR_UNSUPPORTED
DDERR_NOCOLORKEYHW
DDERR_NOCOLORKEY

GetDC

Creates a handle to a device context for the DirectDrawSurface. After calling this function, the DirectDrawSurface object will be locked and should be unlocked with a call to **ReleaseDC** as soon as possible:

```
HRESULT GetDC(  
    LPDIRECTDRAWSURFACE lpDirectDrawSurface,  
    HDC FAR *lphDC)
```

Parameters:

lpDDDirectDrawSurface

Points to the DirectDrawSurface object.

lphDC

Points to an HDC that will receive the device context for the DirectDrawSurface object.

Return values:

DD_OK
DDERR_INVALIDOBJECT
DDERR_INVALIDPARAMS
DDERR_SURFACELOST
DDERR_WASSTILLDRAWING
DDERR_GENERIC
DDERR_UNSUPPORTED

GetFlipStatus

Returns the flip status of the DirectDrawSurface object. Returns **DD_OK** or **DDERR_WASSTILLDRAWING** if the flip is in progress:

```
HRESULT GetFlipStatus(  
    LPDIRECTDRAWSURFACE lpDDSurface,  
    DWORD dwFlags)
```

Parameters:

`lpDDSurface`

Points to the DirectDrawSurface object.

`dwFlags:`

`DDGFS_CANFLIP`

Returns **DD_OK** if the flip can be completed.

`DDGFS_ISFLIPDONE`

Returns **DD_OK** if the last flip on this surface has completed.

Return values:

```
DD_OK  
DDERR_INVALIDPARAMS  
DDERR_INVALIDOBJECT  
DDERR_SURFACELOST  
DDERR_UNSUPPORTED  
DDERR_WASSTILLDRAWING  
DDERR_SURFACEBUSY
```


GetOverlayPosition

Returns the display coordinated of DirectDrawSurface object (DDSCAP_OVERLAY):

```
HRESULT GetOverlayPosition(  
    LPDIRECTDRAWSURFACE lpDDSurface,  
    LPLONG lpIX,  
    LPLONG lpIY)
```

Parameters:

lpDDSurface

Points to the overlay DirectDrawSurface object.

lpIX

Starting x position.

lpIY

Starting y position.

Return values:

```
DD_OK  
DDERR_INVALIDPARAMS  
DDERR_INVALIDOBJECT  
DDERR_SURFACELOST  
DDERR_GENERIC  
DDERR_NOTAOVERLAYSURFACE  
DDERR_NOOVERLAYDEST  
DDERR_OVERLAYNOTVISIBLE  
DDERR_INVALIDPOSITION
```

GetPalette

Returns the DirectDrawPalette object associated with the DirectDrawSurface object.

```
HRESULT GetPalette(  
    LPDIRECTDRAWSURFACE lpDDSurface,  
    LPLPDIRECTDRAWPALETTE lpLPDDPalette)
```

Parameters:

lpDDSurface

Points to the DirectDrawSurface object.

lpLPDDPalette

Points to a pointer to the DirectDrawPalette object. Will be NULL if no palette is associated with the DirectDrawSurface object.

Return values:

```
DD_OK  
DDERR_INVALIDOBJECT  
DDERR_INVALIDPARAMS  
DDERR_SURFACELOST  
DDERR_UNSUPPORTED  
DDERR_GENERIC  
DDERR_NOEXCLUSIVEMODE  
DDERR_NOPALETTEATTACHED
```

GetPixelFormat

Return the color capabilities and the pixel format of the DirectDrawSurface object:

```
HRESULT GetPixelFormat(  
    LPDIRECTDRAWSURFACE lpDDSurface,  
    LPDDPIXELFORMAT lpDDPixelFormat)
```

Parameters:

1pDDSurface

Points to the DirectDrawSurface object.

1pDDPixelFormat

Points to a **DDPIXELFORMAT** structure that will receive the pixel format information of the DirectDrawSurface object.

Return values:

DD_OK
DDERR_INVALIDOBJECT
DDERR_INVALIDPARAMS

GetSurfaceDesc

Returns a surface description of the DirectDrawSurface object:

```
HRESULT GetSurfaceDesc(  
    LPDIRECTDRAWSURFACE 1pDDSurface,  
    LPDDSURFACEDESC 1pDDSurfaceDesc)
```

Parameters:

1pDDSurface

Points to the DirectDrawSurface object.

1pDDSurfaceDesc

Points to a **DDSURFACEDESC** structure that will receive the surface description information of the DirectDrawSurface object.

Return values:

DD_OK
DDERR_INVALIDOBJECT
DDERR_INVALIDPARAMS

Initialize

Part of the COM interface, **Initialize** is called when a DirectDrawSurface object is created:

```
HRESULT Initialize(  
    LPDIRECTDRAWSURFACE lpDDSurface,  
    LPDIRECTDRAW lpDD,  
    LPDDSURFACEDESC lpDDSurfaceDesc )
```

Parameters:

lpDDSurface

Points to the DirectDrawSurface object.

lpDD

Points to the DirectDraw object.

lpDDSurfaceDesc

Points to a DDSURFACEDESC structure.

Return values:

DDERR_ALREADYINITIALIZED

IsLost

Determines whether surface memory associated with the DirectDrawSurface object has been “lost.” Will return **DD_OK** if the surface memory exists or **DDERR_SURFACELOST** if the surface memory is gone:

```
HRESULT IsLost(  
    LPDIRECTDRAWSURFACE lpDDSurface)
```

Parameters:

lpDDSurface

Points to the DirectDrawSurface object.

Return values:

```
DD_OK  
DDERR_INVALIDOBJECT  
DDERR_INVALIDPARAMS  
DDERR_SURFACELOST
```

Lock

Obtains a valid pointer to the DirectDraw object's surface memory. Calling application should **Unlock** the DirectDrawSurface object as soon as possible. DirectDraw Blitting cannot take place while a surface is locked:

```
HRESULT Lock(  
    LPDIRECTDRAWSURFACE lpDDSurface,  
    LPRECT lpDestRect,  
    LPDDSURFACEDESC lpDDSurfaceDesc,  
    DWORD dwFlags,  
    HANDLE hEvent )
```

Parameters:

lpDDSurface

Points to the DirectDrawSurface object.

lpDestRect

If NULL, then the entire surface memory zone is considered "locked." Otherwise, **lpDestRect** points to a RECT structure that defines a region within the drawing surface to lock.

lpDDSurfaceDesc

Points to a **DDSURFACEDESC** structure that will be filled in with the surface description.

dwFlags

DDLOCK_SURFACEMEMORYPTR

This is the default. **Lock** will return pointer to the surface memory associated with the DirectDraw surface object. The pointer will point to the top of **lpDDSurfaceRect** or the top of the surface if **lpDDSurfaceRect** is NULL.

DDLOCK_EVENT

Trigger an event when the surface pointer is locked. Multiple events will be triggered in the order received.

DDLOCK_WAIT

Ignore a **DDERR_SURFACEBUSY** condition. Wait until a valid surface pointer can be returned.

hEvent

Handle to an event that should be triggered when the surface is ready to be locked.

Return values:

DD_OK
DDERR_INVALIDPARAMS
DDERR_INVALIDOBJECT
DDERR_SURFACEBUSY
DDERR_SURFACELOST
DDERR_WASSTILLDRAWING
DDERR_OUTOFMEMORY

QueryInterface

Part of the IUnknown interface, QueryInterface allows a COM object to increase its capabilities while maintaining binary compatibility with existing client programs. For additional information consult the Microsoft Corp. White Paper, "The Component Object Model Specification":

```
HRESULT QueryInterface(  
LPDIRECTDRAW_SURFACE lpDDSurface,  
REFIID riid,  
LPVOID FAR *ppvObj)
```

Parameters:

lpDDSurface

Points to the **DIRECTDRAW_SURFACE** structure representing the DirectDrawSurface.

riid

Points to a UUID (Universally Unique Identifier).

ppvObj

Points to a pointer that will be filled with the interface pointer if the query is successful.

Return values:

```
DD_OK  
DDERR_INVALIDOBJECT  
DDERR_INVALIDPARAMS  
DDERR_OUTOFMEMORY
```

Release

Part of the IUnknown interface, **Release** decreases the reference count of the DirectDrawSurface object:

```
DWORD Release(  
    LPDIRECTDRAWSURFACE lpDDSurface)
```

Parameters:

lpDDSurface

Points to the DIRECTDRAWSURFACE structure representing the DirectDraw Surface.

Return values:

The reference count of the object or zero.

ReleaseDC

Releases the device context (obtained with **GetDC**) associated with the DirectDrawSurface object:

```
HRESULT ReleaseDC(  
    LPDIRECTDRAWSURFACE lpDDSurface,  
    HDC hDC)
```

Parameters:

lpDDSurface

Points to the DirectDrawSurface object.

hDC

The HDC of the DirectDrawSurface object.

Return values:

DD_OK
DDERR_INVALIDPARAMS
DDERR_INVALIDOBJECT
DDERR_SURFACELOST
DDERR_GENERIC
DDERR_UNSUPPORTED

Restore

Restores the surface memory associated with the DirectDrawSurface object:

```
HRESULT Restore(  
    LPDIRECTDRAWSURFACE lpDDSurface)
```

Parameters:

lpDDSurface

Points to the DirectDrawSurface object.

Return values:

DD_OK
DDERR_INVALIDPARAMS
DDERR_IMPLICITLYCREATED
DDERR_INVALIDOBJECT
DDERR_WRONGMODE
DDERR_OUTOFMEMORY
DDERR_NOEXCLUSIVEMODE
DDERR_GENERIC
DDERR_INCOMPATIBLEPRIMARY
DDERR_UNSUPPORTED

SetClipper

Attaches a DirectDrawClipper object to the DirectDrawSurface object:

```
HRESULT SetClipper(  
    LPDIRECTDRAWSURFACE lpDDSurface,  
    LPDIRECTDRAWCLIPPER lpDDClipper)
```

Parameters:

lpDDSurface

Points to the DirectDrawSurface object.

lpDDClipper

Points to the DirectDrawClipper object. If NULL, any previously attached clippers will be detached from the DirectDrawSurface object.

Return values:

```
DD_OK  
DDERR_INVALIDPARAMS  
DDERR_INVALIDOBJECT  
DDERR_NOCLIPPERATTACHED
```

SetColorKey

Attaches a color key the DirectDrawSurface object:

```
HRESULT SetColorKey(  
    LPDIRECTDRAWSURFACE lpDDSurface,  
    DWORD dwFlags,  
    LPDDCOLORKEY lpDDColorKey)
```

Parameters:

lpDDSurface

Points to the DirectDrawSurface object.

dwFlags

Determines the type of color key to apply:

DDCKEY_COLORSPACE
DDCKEY_DESTBLT
DDCKEY_DESTOVERLAY
DDCKEY_SRCBLT
DDCKEY_SRCOVERLAY

lpDDColorKey

Points to the **DDCOLORKEY** structure that will be applied to the DirectDraw Surface object.

Return values:

DD_OK
DDERR_NOOVERLAYHW
DDERR_COLORKEYDRIVERWIDE
DDERR_NODESTCLRKEYHW
DDERR_NOSRCCLRKEYHW
DDERR_INVALIDPARAMS
DDERR_INVALIDOBJECT
DDERR_SURFACELOST
DDERR_UNSUPPORTED
DDERR_WASSTILLDRAWING
DDERR_GENERIC
DDERR_NOTAOVERLAYSURFACE

SetOverlayPosition

Sets the display position of the overlay DirectDrawSurface object:

```
HRESULT SetOverlayPosition(  
    LPDIRECTDRAWSURFACE lpDDSurface,  
    LONG lX,  
    LONG lY)
```

Parameters:

lpDDSurface

Points to the overlay DirectDrawSurface object.

IX

Starting x position.

IY

Starting y position.

Return values:

DD_OK
DDERR_INVALIDPARAMS
DDERR_INVALIDOBJECT
DDERR_SURFACELOST
DDERR_GENERIC
DDERR_UNSUPPORTED

SetPalette

Attaches a DirectDrawPalette object to the DirectDrawSurface object:

```
HRESULT SetPalette(  
    LPDIRECTDRAWSURFACE lpDDSurface,  
    LPDIRECTDRAWPALETTE lpDDPalette)
```

Parameters:

lpDDSurface

Points to the DirectDrawSurface object.

lpDDPalette

Points to the DirectDrawPalette object.

Return values:

```
DD_OK  
DDERR_INVALIDOBJECT  
DDERR_NOEXCLUSIVEMODE  
DDERR_NOT8BITCOLOR  
DDERR_UNSUPPORTED  
DDERR_GENERIC  
DDERR_INVALIDPARAMS  
DDERR_NOPALETTEATTACHED  
DDERR_NOPALETTEHW  
DDERR_SURFACELOST
```

Unlock

Unlocks a previously locked DirectDrawSurface object:

```
HRESULT Unlock(  
    LPDIRECTDRAWSURFACE lpDDSurface,  
    LPVOID lpSurfaceData)
```

Parameters:

lpDDSurface

Points to the DirectDrawSurface object.

lpSurfaceData

Pointer that was returned by **Lock**. Since it is possible to call **Lock** multiple times for the same surface with different destination rectangles, this pointer is used to tie the **Lock** and **Unlock** calls together. Can be NULL if multiple **Lock/Unlocks** aren't an issue.

Return values:

```
DD_OK  
DDERR_INVALIDOBJECT  
DDERR_INVALIDPARAMS  
DDERR_SURFACELOST  
DDERR_NOTLOCKED  
DDERR_GENERIC  
DDERR_INVALIDRECT
```

UpdateOverlay

Updates the position and/or the visual appearance of the overlay DirectDraw Surface object:

```
HRESULT UpdateOverlay(  
    LPDIRECTDRAWSURFACE lpDDSrcSurface,  
    LPRECT lpSrcRect,  
    LPDIRECTDRAWSURFACE lpDDDestSurface,  
    LPRECT lpDestRect,  
    DWORD dwFlags,  
    LPDDOVERLAYFX lpDDOverlayFx)
```

Parameters:

`lpDDSurface`

Points to the overlay DirectDrawSurface object representing the source of the overlay operation.

`lpSrcRect`

Points to a RECT structure that defines size and position of the region on the source surface which is being used as the overlay.

`lpDDDestSurface`

Points to the overlay DirectDrawSurface object representing the source of the overlay operation.

`lpDestRect`

Points to a RECT structure that defines the size and position of the region on the destination surface which the overlay should be moved to.

`dwFlags:`

`DDOVER_ALPHADEST`

Use the alpha-surface attached to the destination surface as the alpha-channel for the blt operation the alpha-channel for the destination overlay.

DDOVER_ALPHADESTCONSTOVERRIDE

Use the **DDOVERLAYFX.dwConstAlphaDest** field as alpha-channel for the destination overlay.

DDOVER_ALPHADESTNEG

The destination alpha-channel surface becomes more transparent as alpha-values increase.

DDOVER_ALPHADESTSURFACEOVERRIDE

Use the **DDOVERLAYFX.lpDDSAlphaDest** field as alpha-channel for the destination overlay.

DDOVER_ALPHAEDGEBLEND

Use the **dwAlphaEdgeBlend** field in the **DDBLTFX** structure as the alpha-channel for the edges of the image that closely match the color key values.

DDOVER_ALPHASRC

Use the alpha-surface attached to the source surface as the alpha-channel for the blt operation the alpha-channel for the source overlay.

DDOVER_ALPHASRCCONSTOVERRIDE

Use the **DDOVERLAYFX.dwConstAlphaSrc** field as alpha-channel for the destination overlay.

DDOVER_ALPHASRCNEG

The source alpha-channel surface becomes more transparent as alpha-values increase.

DDOVER_ALPHASRCSURFACEOVERRIDE

Use the **DDOVERLAYFX.lpDDSAphaSrc** field as alpha-channel for the destination overlay.

DDOVER_HARDWAREONLY

The requested operation must be performed by overlay hardware. If neither this flag nor the **DDOVER_HARDWAREONLY** flag is set, DirectDraw tries to use hardware first and then falls back to software emulation if possible. NOTE: May not be defined in the current version of DDRAW.H

DDOVER_HIDE

Hide the overlay surface.

DDOVER_KEYDEST

Perform operation using the destination color key.

DDOVER_KEYDESTOVERRIDE

Perform operation using the **DDOVERLAYFX.dckDestColorkey** field as the destination color key.

DDOVER_KEYSRC

Perform operation using the source color key.

DDOVER_KEYSRCOVERRIDE

Perform operation using the **DDOVERLAYFX.dckSrcColorkey** field as the source color key.

DDOVER_SHOW

Show the overlay surface.

DDOVER_SOFTWAREONLY

The requested operation must be performed by overlay software emulation. If neither this flag nor the **DDOVER_HARDWAREONLY** flag is set DirectDraw tries to use hardware first and then falls back to software emulation if possible.

DDOVER_ZORDER

Perform operation using the **DDOVERLAYFX.dwZOrderFlags** field as the Z order for the overlay. The **lpDDSRelative** field will be used if the **dwZOrderFlags** field is set to either **DDOVERZ_INSERTINBACKOF** or **DDOVERZ_INSERTINFRONTOF**.

lpDDOverlayFx

Points to a **DDOVERLAYFX** structure.

Return values:

DD_OK
DDERR_SURFACELOST
DDERR_INVALIDPARAMS
DDERR_INVALIDOBJECT
DDERR_INVALIDRECT
DDERR_HEIGHTALIGN
DDERR_XALIGN
DDERR_UNSUPPORTED
DDERR_HEIGHTALIGN
DDERR_NOSTRETCHHW
DDERR_GENERIC
DDERR_NOTAOVERLAYSURFACE

UpdateOverlayDisplay

Repaints the overlay surface based on the dirty rectangle lists all the active overlays surfaces. For software emulated overlay surfaces only:

```
HRESULT UpdateOverlayDisplay(  
    LPDIRECTDRAWSURFACE lpDDSurface,  
    DWORD dwFlags)
```

Parameters:

lpDDSurface

Points to the overlay DirectDrawSurface object.

dwFlags:

DDOVER_REFRESHDIRTYRECTS

Repaint overlays using a dirty rectangle list.

DDOVER_REFRESHALL

Repaint all overlay surfaces.

Return values:

DD_OK
DDERR_INVALIDPARAMS
DDERR_INVALIDOBJECT
DDERR_UNSUPPORTED

UpdateOverlayZOrder

Update overlay surfaces using a pseudo Z order. All positions are relative to other overlays:

```
HRESULT UpdateOverlayZOrder(  
    LPDIRECTDRAWSURFACE lpDDSurface,  
    DWORD dwFlags,  
    LPDIRECTDRAWSURFACE lpDDSReference)
```

Parameters:

lpDDSurface

Points to the overlay DirectDrawSurface object.

dwFlags:

DDOVERZ_INSERTINBACKOF

Place overlay behind the reference overlay.

DDOVERZ_INSERTINFRONTOF

Place overlay in front of the reference overlay.

DDOVERZ_MOVEBACKWARD

Move overlay backwards one slot in the overlay list.

DDOVERZ_MOVEFORWARD

Move overlay forward one slot in the overlay list.

DDOVERZ_SENDBACK

Move overlay to the back of the list.

DDOVERZ_SENDFRONT

Move overlay to the front of the list.

lpDDSReference

Points to an overlay DirectDrawSurface object that will be considered the reference overlay for Z ordering. Used only for **DDOVERZ_INSERTINBACKOF** and **DDOVERZ_INSERTINFRONTOF** flags.

Return values:

DD_OK

DDERR_INVALIDOBJECT

DDERR_INVALIDPARAMS

DDERR_NOTAOVERLAYSURFACE

DIRECTDRAW STRUCTURES

DDBLTBATCH

```
typedef struct _DDBLTBATCH {
    LPRECT          lpDest;
    LPDIRECTDRAWSURFACE lpDDSrc;
    LPRECT          lpSrc;
    DWORD           dwFlags;
    LPDDBLTFX       lpDDBltFx;
} DDBLTBATCH, FAR *LPDDBLTBATCH;
```

`lpDest`

Pointer to a RECT structure that defines the destination size and position of the blt operation.

`lpDDSrc`

Points to source DirectDrawSurface object for the blt operation.

`lpSrc`

Pointer to a RECT structure that defines the source size and position of the blt operation.

`dwFlags`

Specifies an operation from the following list:

- **DDBLT_ALPHADEST.** Use the destination surface as the alpha-channel for the blt operation.
- **DDBLT_ALPHADESTCONSTOVERRIDE.** Use the **dwConstAlphaDest** field in **DDBLTFX** the alpha-channel override for the blt operation.
- **DDBLT_ALPHADESTNEG.** The destination alpha-channel surface becomes more transparent as alpha-values increase.
- **DDBLT_ALPHADESTSURFACEOVERRIDE.** Use the **lpDDSAAlphaDest** field in **DDBLTFX** the alpha-channel override for the blt operation.

- **DDBLT_ALPHAEDGEBLEND.** Use the **dwAlphaEdgeBlend** field in the **DDBLTFX** structure as the alpha-channel for the edges of the image that closely match the color key values.
- **DDBLT_ALPHASRC.** Use the source surface as the alpha-channel for the blt operation.
- **DDBLT_ALPHASRCCONSTOVERRIDE.** Use the **dwConstAlphaSrc** field in **DDBLTFX** the alpha-channel override for the blt operation.
- **DDBLT_ALPHASRCNEG.** The source alpha-channel surface becomes more transparent as alpha-values increase.
- **DDBLT_ALPHASRCSURFACEOVERRIDE.** Use the **lpDDSAAlphaSrc** field in **DDBLTFX** the alpha-channel override for the blt operation.
- **DDBLT_ASYNC.** Perform the blt asynchronously using the hardware FIFO. No FIFO FULL check is performed.
- **DDBLT_COLORFILL.** Perform a color fill on the destination RECT operation the value in the **DDBLTFX.dwFillColor** field as the color fill value.
- **DDBLT_DDFX.** Specify a blt effects operation using the **DDBLTFX.dwDDFX** field.
- **DDBLT_DDROPS.** Specify a non-defined ROPS using the **DDBLTFX.dwDDROPS** field.
- **DDBLT_KEYDEST.** Perform blt operation using the destination color key.
- **DDBLT_KEYDESTOVERRIDE.** Perform blt operation using the **DDBLTFX.dckDestColorkey** field as the destination color key.
- **DDBLT_KEYSRC.** Perform blt operation using the source color key.
- **DDBLT_KEYSRCOVERRIDE.** Perform blt operation using the **DDBLTFX.dckSrcColorkey** field as the source color key.
- **DDBLT_ROP.** Specify a Win32 API defined ROPS using the **DDBLTFX.dwROP** field.
- **DDBLT_ROTATIONANGLE.** Perform the blt operation using **DDBLTFX.dwRotationAngle** field as rotation angle in increments of 1/100th of a degree.
- **DDBLT_ZBUFFER.** Perform a Z buffered blt operation using Z buffers attached to the source and destination drawing surfaces. The **DDBLTFX.dwZBufferOpCode** specifies the Z buffer opcode.

- **DDBLT_ZBUFFERDESTCONSTOVERRIDE**. Perform a Z buffered blt operation using the **DDBLTFX.dwZBufferOpCode** and the **DDBLTFX.dwConstDestZ** as the Z buffer opcode and the Z buffer for the destination surface.
- **DDBLT_ZBUFFERDESTOVERRIDE**. Perform a Z buffered blt operation using the **DDBLTFX.dwZBufferOpCode** and the **DDBLTFX.lpDDSDestZBuffer** as the Z buffer opcode and the Z buffer for the destination surface.
- **DDBLT_ZBUFFERSRCCONSTOVERRIDE**. Perform a Z buffered blt operation using the **DDBLTFX.dwZBufferOpCode** and the **DDBLTFX.dwConstSrcZ** as the Z buffer opcode and the Z buffer for the source surface.
- **DDBLT_ZBUFFERSRCOVERRIDE**. Perform a Z buffered blt operation using the **DDBLTFX.dwZBufferOpCode** and the **DDBLTFX.lpDDSSrcZBuffer** as the Z buffer opcode and the Z buffer for the source surface.

lpDDBlTfx

Points to a **DDBLTFX** structure.

```
typedef struct _DDBLTFX{
    DWORD dwSize;
    DWORD dwDDFX;
    DWORD dwROP;
    DWORD dwDDROP;
    DWORD dwRotationAngle;
    DWORD dwZBufferOpCode;
    DWORD dwZBufferLow;
    DWORD dwZBufferHigh;
    DWORD dwZBufferBaseDest;
    DWORD dwZDestConstBitDepth;
    union
    {
        DWORD dwZDestConst;
        LPDIRECTDRAWSURFACE lpDDSZBufferDest;
    };
    DWORD dwZSrcConstBitDepth;
    union
    {
        DWORD dwZSrcConst;
        LPDIRECTDRAWSURFACE lpDDSZBufferSrc;
    };
    DWORD dwAlphaEdgeBlendBitDepth;
```

```

DWORD dwAlphaEdgeBlend;
DWORD dwReserved;
DWORD dwAlphaDestConstBitDepth;
union
{
    DWORD          wAlphaDestConst;
    LPDIRECTDRAW SURFACE lpDDSAAlphaDest;
};
DWORD dwAlphaSrcConstBitDepth;
union
{
    DWORD          dwAlphaSrcConst;
    LPDIRECTDRAW SURFACE lpDDSAAlphaSrc;
};
union
{
    WORD          dwFillColor;
    LPDIRECTDRAW SURFACE lpDDSPattern;
};
DDCOLORKEY dckDestColorkey;
DDCOLORKEY dckSrcColorkey;
} DDBLTFX, FAR* LPDDBLTFX;

```

dwSize

Must contain the size of the **DDBLTFX** structure.

dwDDFX

Specifies an FX operation from the following list:

- **DDBLTFX_ARITHSTRETCHY**. Perform blt using y-axis arithmetic stretching.
- **DDBLTFX_MIRRORLEFTRIGHT**. Perform blt, mirroring pixels left to right.
- **DDBLTFX_MIRRORUPDOWN**. Perform blt, mirroring pixels up and down.
- **DDBLTFX_NOTEARING**. Unknown operation.
- **DDBLTFX_ROTATE180**. Perform blt, rotating surface pixels 180°.
- **DDBLTFX_ROTATE270**. Perform blt, rotating surface pixels 270°.
- **DDBLTFX_ROTATE90**. Perform blt, rotating surface pixels 90°.

- **DDBLTFX_ZBUFFERRANGE**. Perform Z blt operation using **dwZBufferLow** and **dwZBufferHigh** as low and high Z ranges for the source surface.
- **DDBLTFX_ZBUFFERBASEDEST**. Perform Z blt operation adding **dwZBufferBaseDest** to source Z values before comparing it with the destination Z values.

dwROP

Win32 ROPS codes.

dwDDROP

User defined ROPS.

dwRotationAngle

Rotation angle for the blt operation.

dwZBufferOpCode

ZBuffer compare values.

dwZBufferLow

Limit of Z buffer low range.

dwZBufferHigh

Limit of Z buffer high range.

dwZBufferBaseDest

Z Destination base value.

dwZDestConstBitDepth

Bit depth, specifies Z constant for destination.

dwZDestConst

Constant Z buffer for destination.

lpDDSZBufferDest

Z buffer destination surface.

dwZSrcConstBitDepth

Bit depth, specifies Z constant for source.

dwZSrcConst

Constant Z buffer for source.

lpDDSZBufferSrc

Z buffer source surface.

dwAlphaEdgeBlendBitDepth

Bit depth, constant for alpha-edge blend.

dwAlphaEdgeBlend

alpha-value for edge blending.

dwReserved

Reserved for future use.

dwAlphaDestConstBitDepth

Bit depth, specifies alpha-constant for destination.

dwAlphaDestConst

alpha-channel constant.

lpDDSA1phaDest

alpha-channel destination surface.

dwAlphaSrcConstBitDepth

Bit depth, specifies alpha-constant for source.

dwAlphaSrcConst

alpha-channel constant.

lpDDSA1phaSrc

alpha-channel source surface.

dwFillColor

Fill color for RGB or palettized fill operations.

lpDDSPattern

Surface to use as a tile pattern.

dckDestColorkey

Destination color key override structure.

dckSrcColorkey

Source color key override structure.

DDCAPS

```
typedef struct _DDCAPS{  
    DWORD dwSize;  
    DWORD dwCaps;  
    DWORD dwCaps2;
```

```

DWORD dwKeyCaps;
DWORD dwFXCaps;
DWORD dwFXAlphaCaps;
DWORD dwPalCaps;
DWORD dwSVCaps;
DWORD dwAlphaBitConstBitDepths;
DWORD dwAlphaBitPixelBitDepths;
DWORD dwAlphaBitSurfaceBitDepths;
DWORD dwAlphaOverlayConstBitDepths;
DWORD dwAlphaOverlayPixelBitDepths;
DWORD dwAlphaOverlaySurfaceBitDepths;
DWORD dwZBufferBitDepths;
DWORD dwVidMemTotal;
DWORD dwVidMemFree;
DWORD dwMaxVisibleOverlays;
DWORD dwCurrVisibleOverlays;
DWORD dwNumFourCCCodes;
DWORD dwAlignBoundarySrc;
DWORD dwAlignSizeSrc;
DWORD dwAlignBoundaryDest;
DWORD dwAlignSizeDest;
DWORD dwAlignStrideAlign;
DWORD dwRops[DD_ROP_SPACE];
DDSCAPS ddsCaps;
DWORD dwMinOverlayStretch;
DWORD dwMaxOverlayStretch;
DWORD dwMinLiveVideoStretch;
DWORD dwMaxLiveVideoStretch;
DWORD dwMinHwCodecStretch;
DWORD dwMaxHwCodecStretch;
DWORD dwReserved1;
DWORD dwReserved2;
DWORD dwReserved3;
} DDSCAPS, FAR* LPDDSCAPS;

```

dwSize

Must contain the size of the **DDSCAPS** structure.

dwCaps

Driver specific capabilities. From the following list:

- **DDSCAPS_3D**. Video hardware is capable of 3D operations.
- **DDSCAPS_ALIGNBOUNDARYDEST**. Only source rectangles with an X alignment of **DIRECTDRAWCAPS.dwAlignBoundaryDest** are allowed.

- **DDCAPS_ALIGNSIZEDEST.** Only destination rectangles with an *x* alignment of **DIRECTDRAWCAPS.dwAlignBoundaryDest** are allowed.
- **DDCAPS_ALIGNBOUNDARYSRC.** Only source rectangles with an *x* alignment of **DIRECTDRAWCAPS.dwAlignBoundarySrc** are allowed.
- **DDCAPS_ALIGNSIZESRC.** Only source rectangles with a width **DIRECTDRAWCAPS.dwAlignSizeSrc** multiples are allowed.
- **DDCAPS_ALIGNSTRIDE.** Surfaces can only be created with a width that matches the value in **dwAlignStrideAlign**.
- **DDCAPS_BANKSWITCHED.** The display hardware must utilize bank switching to randomly access video memory.
- **DDCAPS_BLT.** Display hardware has a bltter chip.
- **DDCAPS_BLTCOLORFILL.** The display hardware can perform a color fill operation.
- **DDCAPS_BLTQUEUE.** The Display hardware is capable of asynchronous blts.
- **DDCAPS_BLTFOURCC.** The display hardware can perform runtime color conversions.
- **DDCAPS_BLTSTRETCH.** The display hardware can perform stretch blts.
- **DDCAPS_GDI.** The display hardware is also used by the GDI.
- **DDCAPS_OVERLAY.** The display hardware can perform overlay operations.
- **DDCAPS_OVERLAYCANTCLIP.** The display hardware can perform overlay operations but can't clip overlays.
- **DDCAPS_OVERLAYFOURCC.** The display hardware can perform color conversion during overlay operations.
- **DDCAPS_OVERLAYSTRETCH.** The display hardware can perform overlay operations with stretching.
- **DDCAPS_PALETTE.** The display hardware can support multiple palettes.
- **DDCAPS_PALETTECANVSYNC.** The display hardware can update the hardware palette during the VBI.
- **DDCAPS_READSCANLINE.** The display hardware can return the current scan line.

- **DDCAPS_STEREOVIEW.** The display hardware has stereo vision capabilities. **DDSCAPS_PRIMARYSURFACELEFT** can be created.
- **DDCAPS_VBI.** The display hardware can generate vertical blank interrupt.
- **DDCAPS_ZBLTS.** The display hardware can perform Z buffering operations.
- **DDCAPS_ZOVERLAYS.** The display hardware can Z order multiple overlays.
- **DDCAPS_COLORKEY.** *Obsolete* flag.
- **DDCAPS_ALPHA.** The display hardware can perform alpha-channeling.
- **DDCAPS_COLORKEY_HWASSIST.** The display hardware can implement a color key in hardware.
- **DDCAPS_NOHARDWARE.** The display hardware contains no hardware support.

wCaps2

Additional driver-specific capabilities. From the following list:

- **DDCAPS2_CERTIFIED.** The display hardware has been certified by Microsoft Corp.
- **dwCKeyCaps.** Color key capabilities. From the following list:
- **DDCKEYCAPS_DESTBLT.** The display hardware supports RGB destination color keying.
- **DDCKEYCAPS_DESTBLTCLRSPACE.** The display hardware supports RGB destination color space keying.
- **DDCKEYCAPS_DESTBLTCLRSPACEYUV.** The display hardware supports YUV destination color space keying.
- **DDCKEYCAPS_DESTBLTYUV.** The display hardware supports YUV destination color keying.
- **DDCKEYCAPS_DESTOVERLAY.** The display hardware supports RGB destination overlay color keying.
- **DDCKEYCAPS_DESTOVERLAYCLRSPACE.** The display hardware supports RGB destination overlay color space keying.
- **DDCKEYCAPS_DESTOVERLAYCLRSPACEYUV.** The display hardware supports YUV destination overlay color space keying.

- **DDCKEYCAPS_DESTOVERLAYONEACTIVE.** The display hardware supports only one active destination overlay color key.
- **DDCKEYCAPS_DESTOVERLAYYUV.** The display hardware supports YUV destination overlay color keying.
- **DDCKEYCAPS_SRCBLT.** The display hardware supports RGB source color keying.
- **DDCKEYCAPS_SRCBLTCLRSPACE.** The display hardware supports RGB destination color space keying.
- **DDCKEYCAPS_SRCBLTCLRSPACEYUV.** The display hardware supports YUV destination color space keying.
- **DDCKEYCAPS_SRCBLTYUV.** The display hardware supports YUV destination color keying.
- **DDCKEYCAPS_SRCOVERLAY.** The display hardware supports RGB source overlay color keying.
- **DDCKEYCAPS_SRCOVERLAYCLRSPACE.** The display hardware supports RGB source overlay color space keying.
- **DDCKEYCAPS_SRCOVERLAYCLRSPACEYUV.** The display hardware supports YUV source overlay color space keying.
- **DDCKEYCAPS_SRCOVERLAYONEACTIVE.** The display hardware supports only one active source overlay color key.
- **DDCKEYCAPS_SRCOVERLAYYUV.** The display hardware supports YUV source overlay color keying.

dwFXCaps

Driver-specific stretching and effects capabilities. From the following list:

- **DDFXCAPS_BLITARITHSTRETCHY.** Can support blt operations using arithmetic operations to stretch and shrink surface pixels along the y-axis.
- **DDFXCAPS_BLITARITHSTRETCHYN.** Can perform blt operations using arithmetic operations to stretch and shrink surface pixels along the y-axis. Only works for x1, x2, etc.
- **DDFXCAPS_BLITMIRRORLEFTRIGHT.** Can perform blt mirroring pixels left to right.

- **DDFXCAPS_BLITMIRRORUPDOWN.** Can perform blt mirroring pixels top to bottom.
- **DDFXCAPS_BLITROTATION.** Can perform blt using an arbitrary rotational value.
- **DDFXCAPS_BLITROTATION90.** Can perform blt rotating pixels 90°.
- **DDFXCAPS_BLITSHRINKX.** Can perform blt shrinking pixels along the *x*-axis.
- **DDFXCAPS_BLITSHRINKXN.** Can perform blt using integer shrinking along the *x*-axis.
- **DDFXCAPS_BLITSHRINKY.** Can perform blt shrinking pixels along the *y*-axis.
- **DDFXCAPS_BLITSHRINKYN.** Can perform blt shrinking by factors (1*x*,2*x*,) along the *y*-axis.
- **DDFXCAPS_BLITSTRETCHX.** Can perform blt using arbitrary stretching along the *x*-axis.
- **DDFXCAPS_BLITSTRETCHXN.** Can perform blt stretching by integer factors (1*x*,2*x*,) along the *x*-axis.
- **DDFXCAPS_BLITSTRETCHY.** Can perform blt using arbitrary stretching along the *y*-axis.
- **DDFXCAPS_BLITSTRETCHYN.** Can perform blt stretching by integer factors (1*x*,2*x*,) along the *y*-axis.
- **DDFXCAPS_OVERLAYARITHSTRETCHY.** Can perform blt stretching and shrinking using arithmetic operations on overlays along the *y*-axis.
- **DDFXCAPS_OVERLAYARITHSTRETCHYN.** Can perform blt stretching and shrinking using arithmetic operations (*x*1, *x*2, etc.) on overlays along the *y*-axis.
- **DDFXCAPS_OVERLAYSHRINKX.** Can perform blt using arbitrary shrinking for overlays along the *x*-axis.
- **DDFXCAPS_OVERLAYSHRINKXN.** Can perform blt shrinking by integer factors (1*x*,2*x*,) along the *x*-axis.
- **DDFXCAPS_OVERLAYSHRINKY**
- Can perform blt using arbitrary shrinking for overlays along the *y*-axis.

- **DDFXCAPS_OVERLAYSHRINKYN.** Can perform blt shrinking by integer factors (1x,2x,) along the y-axis.
- **DDFXCAPS_OVERLAYSTRETCHX.** Can perform blt using arbitrary stretching for overlays along the x-axis.
- **DDFXCAPS_OVERLAYSTRETCHXN.** Can perform blt stretching by integer factors (1x,2x,) along the x-axis.
- **DDFXCAPS_OVERLAYSTRETCHY.** Can perform blt using arbitrary stretching for overlays along the y-axis.
- **DDFXCAPS_OVERLAYSTRETCHYN.** Can perform blt stretching by integer factors (1x,2x,) along the y-axis.
- **DDFXCAPS_OVERLAYMIRRORLEFTRIGHT.** Can perform blt mirroring overlay pixels left to right.
- **DDFXCAPS_OVERLAYMIRRORUPDOWN.** Can perform blt mirroring overlay pixels top to bottom.

dwFXAlphaCaps

Driver-specific alpha-capabilities. From the following list:

- **DDFXALPHACAPS_BLTALPHAEDGEBLEND.** Can perform blt using alpha-blending around the edge of a source color keyed surface.
- **DDFXALPHACAPS_BLITALPHAPIXELS.** Can perform blt using alpha-information in the pixel format (bpp 1,2,4,or 8). Higher alpha-values equal increased opacity.
- **DDFXALPHACAPS_BLITALPHAPIXELSNEG.** Can perform blt using alpha-information in the pixel format (bpp 1,2,4,or 8). Higher alpha-values equal decreased opacity.
- **DDFXALPHACAPS_BLITALPHASURFACES.** Can perform blt using alpha-surfaces (bpp 1,2,4,or 8). Higher alpha-values equal increased opacity.
- **DDFXALPHACAPS_BLITALPHASURFACESNEG.** Can perform alpha-blit using a NEG override. See the **Blit** function.

- **DDFXALPHACAPS_OVERLAYALPHAEDGEBLEND.** Can perform alpha-blt to overlay using alpha-blending around the edge of a source color keyed surface.
- **DDFXALPHACAPS_OVERLAYALPHAPIXELS.** Can perform alpha-blt to overlay surface.
- **DDFXALPHACAPS_OVERLAYALPHAPIXELSNeg.** Can perform alpha-blt to overlay surface using a NEG override. See the **Blt** function.
- **DDFXALPHACAPS_OVERLAYALPHASURFACES.** For overlay surface operations.
- **DDFXALPHACAPS_OVERLAYALPHASURFACESNEG.** For overlay surface operations

dwPalCaps

Palette capabilities. From the following list:

- **DDPCAPS_4BIT.** The display hardware can realise a 4 bit, double indexed palette.
- **DDPCAPS_8BITENTRIES.** The display can index a 4 bit palette into an existing 8 bit palette.
- **DDPCAPS_8BIT.** The display hardware can generate a 256 color display.
- **DDPCAPS_ALLOW256.** The display hardware will allow all 256 colors to be defined.
- **DDPCAPS_INITIALIZE.** Unknown.
- **DDPCAPS_PRIMARYSURFACE**
- The display hardware allows only one palette, attached to primary surface. Changes to this palette be immediately visible unless **DDPAL_VSYNC** is specified and supported.
- **DDPCAPS_PRIMARYSURFACELEFT.** The display hardware allows only one palette, attached to primary surface left. Changes to this palette be immediately visible unless **DDPAL_VSYNC** is specified and supported.
- **DDPCAPS_VSYNC.** The display hardware can ensure that palette changes will take place during the VBI.

dwSVCaps

Stereo vision capabilities. From the following list:

- DDSVCAPS_ENIGMA
- DDSVCAPS_FLICKER
- DDSVCAPS_REDBLUE
- DDSVCAPS_SPLIT

DDBD_1 = 1 bit per pixel.
DDBD_2 = 2 bits per pixel.
DDBD_4 = 4 bits per pixel.
DDBD_8 = 8 bits per pixel.
DDBD_16 = 16 bits per pixel.
DDBD_24 = 24 bits per pixel.
DDBD_32 = 32 bits per pixel.

dwAlphaBitConstBitDepths
DDBD_2, DDBD_4, or DDBD_8.
dwAlphaBitPixelBitDepths
DDBD_1, DDBD_2, DDBD_4, or DDBD_8.
dwAlphaBitSurfaceBitDepths
DDBD_1, DDBD_2, DDBD_4, or DDBD_8.
dwAlphaOverlayConstBitDepths
DDBD_2, DDBD_4, or DDBD_8.
dwAlphaOverlayPixelBitDepths
DDBD_1, DDBD_2, DDBD_4, or DDBD_8.
dwAlphaOverlaySurfaceBitDepths
DDBD_1, DDBD_2, DDBD_4, or DDBD_8.
dwZBufferBitDepths
DDBD_8, DDBD_16, DDBD_24, or DDBD_32.

dwVidMemTotal

Total amount of video memory contained on the display hardware.

dwVidMemFree

Amount of free video memory at any given time.

`dwMaxVisibleOverlays`

Maximum number of visible overlays allowed.

`dwCurrVisibleOverlays`

Number of currently visible overlays.

`dwNumFourCCCodes`

Number of FOURCC color conversion codes.

`dwAlignBoundarySrc`

Source rectangle alignment.

`dwAlignSizeSrc`

Source rectangle byte size.

`dwAlignBoundaryDest`

Destination rectangle alignment.

`dwAlignSizeDest`

Destination rectangle byte size.

`dwAlignStrideAlign`

Stride alignment.

`dwRops [DD_ROP_SPACE]`

ROPS supported.

ddsCaps

DDSCAPS structure with general capabilities.

dwMinOverlayStretch

Minimum overlay stretch factor multiplied by 1000 (e.g., 1000 == 1.0, 1300 == 1.3).

dwMaxOverlayStretch

Maximum overlay stretch factor multiplied by 1000.

dwMinLiveVideoStretch

Minimum live video stretch factor multiplied by 1000.

dwMaxLiveVideoStretch

Maximum live video stretch factor multiplied by 1000.

dwMinHwCodecStretch

Minimum hardware codec stretch factor multiplied by 1000.

dwMaxHwCodecStretch

Maximum hardware codec stretch factor multiplied by 1000.

dwReserved1

Reserved.

dwReserved2

Reserved.

dwReserved3

Reserved.

DDCOLORKEY

```
typedef struct _DDCOLORKEY{  
    DWORD dwColorSpaceLowValue;  
    DWORD dwColorSpaceHighValue;  
} DDCOLORKEY, FAR* LPDDCOLORKEY;
```

dwColorSpaceLowValue

Specifies the low value for the color-key range as a palette index, RGB, or Y U V value.

dwColorSpaceHighValue

Specifies the high value for the color-key range as a palette index, RGB, or YUV value.

DDMODEDESC

```
typedef struct {  
    DWORD    dwSize;  
    DWORD    dwFlags;  
    DWORD    dwMonitorFrequency;  
    DDSURFACEDESC    dsdSurfaceDesc;  
} DDMODEDESC, FAR* LPDDMODEDESC;
```

dwSize

Must contain the size of the **DDMODEDESC** structure.

dwFlags

Enumeration flags.

dwMonitorFrequency

Frequency of the display monitor in this mode.

dsdSurfaceDesc

Surface description.

DDOVERLAYFX

```
typedef struct _DDOVERLAYFX{
    DWORD dwSize;
    DWORD dwAlphaEdgeBlendBitDepth;
    DWORD dwAlphaEdgeBlend;
    DWORD dwReserved;
    DWORD dwAlphaDestConstBitDepth;
    union
    {
        {
            DWORD dwAlphaDestConst;
            LPDIRECTDRAWSURFACE lpDDSAAlphaDest;
        };
        DWORD dwAlphaSrcConstBitDepth;
        {
            DWORD dwAlphaSrcConst;
            LPDIRECTDRAWSURFACE lpDDSAAlphaSrc;
        };
    };
    DDCOLORKEY dckDestColorkey;
    DDCOLORKEY dckSrcColorkey;
    DWORD dwDDFX;
    DWORD dwFlags;
} DDOVERLAYFX, FAR *LPDDOVERLAYFX;
```

dwSize

Must contain the size of the **DDOVERLAYFX** structure.

dwAlphaEdgeBlendBitDepth

Bit depth for alpha-edge blend.

`dwAlphaEdgeBlend`

Constant for edge blend.

`dwReserved`

Reserved.

`dwAlphaDestConstBitDepth`

Bit depth, specifies alpha-constant for destination surface.

`dwAlphaDestConst`

alpha-channel constant for destination surface.

`lpDDSAAlphaDest`

Points to alpha-surface to use as the destination surface.

`dwAlphaSrcConstBitDepth`

Bit depth, specifies alpha-constant for the source surface.

`dwAlphaSrcConst`

alpha-channel constant for the source surface.

`lpDDSAAlphaSrc`

Points to alpha-surface to use as the source surface.

`dckDestColorkey`

Use as destination color key override.

dckSrcColorkey

Use as source color key override.

dwDDFX

Overlay FX flags. From the following list:

- **DDOVERFX_ARITHSTRETCHY.** Use arithmetic stretching along the y-axis for this overlay.
- **DDOVERFX_MIRRORLEFTRIGHT.** Mirror the overlay left to right.
- **DDOVERFX_MIRRORUPDOWN.** Mirror the overlay top to bottom.

dwFlags

Reserved.

DDPIXELFORMAT

```
typedef struct _DDPIXELFORMAT{
    DWORD dwSize;
    DWORD dwFlags;
    DWORD dwFourCC;
    union
    {
        DWORD dwRGBBitCount;
        DWORD dwYUVBitCount;
        DWORD dwZBufferBitDepth;
        DWORD dwAlphaBitDepth;
    };
    union
    {
        DWORD dwRBitMask;
        DWORD dwYBitMask;
    };
    union
    {
        DWORD dwGBitMask;
```



```

        DWORD dwUBitMask;
    };
    union
    {
        DWORD dwBBitMask;
        DWORD dwVBitMask;
    };
    union
    {
        DWORD dwRGBAlphaBitMask;
        DWORD dwYUVAAlphaBitMask;
    };
} DDPIXELFORMAT, FAR* LPDDPIXELFORMAT;

```

dwSize

Must contain the size of the **DDPIXELFORMAT** structure.

dwFlags

Pixel format flags. From the following list:

- **DDPF_ALPHAPIXELS**. alpha-channel information is contained in the pixel format.
- **DDPF_ALPHA**. The surface is alpha-only.
- **DDPF_FOURCC**. The FourCC code is valid.
- **DDPF_PALETTEINDEXED4**. The surface is 4-bit color indexed.
- **DDPF_PALETTEINDEXED4TO8**. The surface is 4-bit color, indexed to an existing 8-bit palette.
- **DDPF_PALETTEINDEXED8**. The surface is 8-bit.
- **DDPF_RGB**. The surface is RGB.
- **DDPF_COMPRESSED**. The surface can contain compressed pixel data.
- **DDPF_RGBTOYUV**. The surface can convert RGB to YUV.
- **DDPF_YUV**. The surface contains YUV pixels.
- **DDPF_ZBUFFER**. The surface is a Z buffer.

dwFourCC

The FOURCC code.

dwRGBBitCount

RGB bits per pixel (DDBD_4,8,16,24,32).

dwYUVBitCount

YUV bits per pixel (DDBD_4,8,16,24,32).

dwZBufferBitDepth

Z buffer bit depth (DDBD_8,16,24,32).

dwAlphaBitDepth

alpha-channel bit depth (DDBD_1,2,4,8).

dwRBitMask

Mask for RGB red.

dwYBitMask

Mask for YUV Y.

dwGBitMask

Mask for RGB green.

dwUBitMask

Mask for YUV U.

dwBBitMask

Mask for RGB blue.

`dwVBitMask`

Mask for YUV V.

`dwRGBAlphaBitMask`

Mask for alpha-channel.

`dwYUVAAlphaBitMask`

Mask for alpha-channel.

DDSCAPS

```
typedef struct _DDSCAPS{  
    DWORD dwCaps;  
} DDSCAPS, FAR* LPDDSCAPS;
```

`dwCaps`

Indicates the type and capabilities of a surface. From the following list:

- **DDSCAPS_3D**. Indicates that the surface is being used in conjunction with a 3DDDI or Direct3D HAL.
- **DDSCAPS_ALPHA**. The surface contain alpha-information.
- **DDSCAPS_BACKBUFFER**. The surface is a backbuffer.
- **DDSCAPS_COMPLEX**. The surface forms one part of a surface attachment structure.
- **DDSCAPS_FLIP**. The surface forms one part of a page flippable surface attachment structure.
- **DDSCAPS_FRONTBUFFER**. The surface is the front buffer in a page flippable surface attachment structure.
- **DDSCAPS_HWCODEC**. The surface can accept streaming data from the hardware.

- **DDSCAPS_LIVEVIDEO.** The surface can receive live video feed.
- **DDSCAPS_MODEX.** The surface can support ModeX displays at 320 x 200 or 320 x 240.
- **DDSCAPS_OFFSCREENPLAIN.** The surface is a plain off screen surface.
- **DDSCAPS_OWNDC.** The surface can support a long term DC.
- **DDSCAPS_OVERLAY.** The surface is an overlay.
- **DDSCAPS_PALETTE.** The surface can have a unique palette associated with it.
- **DDSCAPS_PRIMARYSURFACE.** The surface is the primary surface.
- **DDSCAPS_PRIMARYSURFACELEFT.** The surface is the primary surface left.
- **DDSCAPS_SYSTEMMEMORY.** The surface memory exists in system memory.
- **DDSCAPS_TEXTUREMAP.** The surface is a 3D texture map.
- **DDSCAPS_VIDEOMEMORY.** The surface memory exists in video memory.
- **DDSCAPS_VISIBLE.** The surface is visible to the user.
- **DDSCAPS_WRITEONLY (read only).** The surface cannot be reliably READ from.
- **DDSCAPS_ZBUFFER.** The surface is a Z buffer.

DDSURFACEDESC

```
typedef struct _DDSURFACEDESC{
    DWORD      dwSize;
    DWORD      dwFlags;
    DWORD      dwHeight;
    DWORD      dwWidth;
    LONG       lpPitch;
    DWORD      dwBackBufferCount;
    DWORD      dwZBufferBitDepth;
    DWORD      dwAlphaBitDepth;
    LPVOID      lpSurface;
    DDCOLORKEY ddckCKDestOverlay;
    DDCOLORKEY ddckCKDestBlt;
    DDCOLORKEY ddckCKSrcOverlay;
}
```

```
DDCOLORKEY ddckCKSrcBlt;
DDPIXELFORMAT ddpfPixelFormat;
DDSCAPS ddsCaps;
} DDSURFACEDESC, FAR* LPDDSURFACEDESC;
```

dwSize

Must contain the size of the **DDSURFACEDESC** structure.

dwFlags

Determines which **DDSURFACEDESC** fields are valid. From the following list:

- **DDSD_DDSCAPS**. **ddsCaps** field is valid.
- **DDSD_HEIGHT**. **dwHeight** field is valid.
- **DDSD_WIDTH**. **dwWidth** field is valid.
- **DDSD_PITCH**. **lpPitch** field is valid.
- **DDSD_BACKBUFFERCOUNT**. **dwBackBufferCount** field is valid.
- **DDSD_ZBUFFERBITDEPTH**. **dwZBufferBitDepth** field is valid.
- **DDSD_ALPHABITDEPTH**. **dwAlphaBitDepth** field is valid.
- **DDSD_LPSURFACE**. **lpSurface** field is valid.
- **DDSD_PIXELFORMAT**. **ddpfPixelFormat** field is valid.
- **DDSD_CKDESTOVERLAY**. **ddckCKDestOverlay** field is valid.
- **DDSD_CKDESTBLT**. **ddckCKDestBlt** field is valid.
- **DDSD_CKSRCOVERLAY**. **ddckCKSrcOverlay** field is valid.
- **DDSD_CKSRGBLT**. **ddckCKSrcBlt** field is valid.
- **DDSD_ALL**. All fields are valid.

dwHeight

Height of input surface.

dwWidth

Width of input surface.

`lpPitch`

Distance to start of next line within surface memory.

`dwBackBufferCount`

Number of back buffers attached to the surface.

`dwZBufferBitDepth`

Z buffer depth.

`dwAlphaBitDepth`

alpha-channel depth.

`lpSurface`

Pointer to the associated surface memory.

`ddckCKDestOverlay`

Overlay color key for destination.

`ddckCKDestBlt`

Color key for destination.

`ddckCKSrcOverlay`

Overlay color key for source.

`ddckCKSrcBlt`

Color key for source.

ddpfPixelFormat

Pixel format description of the surface (**DDPIXELFORMAT**).

ddsCaps

Surface capabilities.

DirectDraw

```

DECLARE_INTERFACE_( IDirectDraw, IUnknown )
{
    /*** IUnknown methods ***/
    STDMETHOD(QueryInterface)(THIS_ REFIID riid, LPVOID FAR * ppvObj)
    PURE;
    STDMETHOD_(ULONG,AddRef)(THIS) PURE;
    STDMETHOD_(ULONG,Release)(THIS) PURE;
    /*** IDirectDraw methods ***/
    STDMETHOD(Compact)(THIS) PURE;
    STDMETHOD(CreateClipper)(THIS_ DWORD, LPDIRECTDRAWCLIPPER FAR*,
        IUnknown FAR * ) PURE;
    STDMETHOD(CreatePalette)(THIS_ DWORD, LPPALETTEENTRY,
        LPDIRECTDRAWPALETTE FAR*, IUnknown FAR * ) PURE;
    STDMETHOD(CreateSurface)(THIS_ LPDDSURFACEDESC,
        LPDIRECTDRAWSURFACE FAR *, IUnknown FAR * ) PURE;
    STDMETHOD(DuplicateSurface)( THIS_ LPDIRECTDRAWSURFACE,
        LPDIRECTDRAWSURFACE FAR * ) PURE;
    STDMETHOD(EnumDisplayModes)( THIS_ DWORD, LPDDSURFACEDESC, LPVOID,
        LPDDENUMMODESCALLBACK ) PURE;
    STDMETHOD(EnumSurfaces)(THIS_ DWORD, LPDDSURFACEDESC,
        LPVOID,LPDDENUMSURFACESCALLBACK ) PURE;
    STDMETHOD(FlipToGDISurface)(THIS) PURE;
    STDMETHOD(GetCaps)( THIS_ LPDDCAPS, LPDDCAPS ) PURE;
    STDMETHOD(GetDisplayMode)( THIS_ LPDDSURFACEDESC ) PURE;
    STDMETHOD(GetFourCCCodes)(THIS_ LPDWORD, LPDWORD ) PURE;
    STDMETHOD(GetGDISurface)(THIS_ LPDIRECTDRAWSURFACE FAR *) PURE;
    STDMETHOD(GetMonitorFrequency)(THIS_ LPDWORD) PURE;
    STDMETHOD(GetScanLine)(THIS_ LPDWORD) PURE;
    STDMETHOD(GetVerticalBlankStatus)(THIS_ LPBOOL ) PURE;
    STDMETHOD(Initialize)(THIS_ GUID FAR *) PURE;
    STDMETHOD(RestoreDisplayMode)(THIS) PURE;
    STDMETHOD(SetCooperativeLevel)(THIS_ HWND, DWORD) PURE;
    STDMETHOD(SetDisplayMode)(THIS_ DWORD, DWORD,DWORD) PURE;
    STDMETHOD(WaitForVerticalBlank)(THIS_ DWORD, HANDLE ) PURE;
};
    
```

DirectDrawClipper

```
DECLARE_INTERFACE_( IDirectDrawClipper, IUnknown )
{
    /*** IUnknown methods ***/
    STDMETHOD(QueryInterface) (THIS_ REFIID riid, LPVOID FAR * ppvObj)
    PURE;
    STDMETHOD_(ULONG,AddRef) (THIS) PURE;
    STDMETHOD_(ULONG,Release) (THIS) PURE;
    /*** IDirectDrawClipper methods ***/
    STDMETHOD(GetClipList)(THIS_ LPRECT, LPRGNDATA, LPDWORD) PURE;
    STDMETHOD(GetHWnd)(THIS_ HWND FAR *) PURE;
    STDMETHOD(Initialize)(THIS_ LPDIRECTDRAW, DWORD) PURE;
    STDMETHOD(IsClipListChanged)(THIS_ BOOL FAR *) PURE;
    STDMETHOD(SetClipList)(THIS_ LPRGNDATA,DWORD) PURE;
    STDMETHOD(SetHWnd)(THIS_ DWORD, HWND ) PURE;
};
```

DirectDrawPalette

```
DECLARE_INTERFACE_( IDirectDrawPalette, IUnknown )
{
    /*** IUnknown methods ***/
    STDMETHOD(QueryInterface) (THIS_ REFIID riid, LPVOID FAR * ppvObj)
    PURE;
    STDMETHOD_(ULONG,AddRef) (THIS) PURE;
    STDMETHOD_(ULONG,Release) (THIS) PURE;
    /*** IDirectDrawPalette methods ***/
    STDMETHOD(GetCaps)(THIS_ LPDWORD) PURE;
    STDMETHOD(GetEntries)(THIS_ DWORD,DWORD,DWORD,LPPALETTEENTRY)
    PURE;
    STDMETHOD(Initialize)(THIS_ LPDIRECTDRAW, DWORD, LPPALETTEENTRY)
    PURE;
    STDMETHOD(SetEntries)(THIS_ DWORD,DWORD,DWORD,LPPALETTEENTRY)
    PURE;
};
```


DirectDrawSurface

```

DECLARE_INTERFACE_( IDirectDrawSurface, IUnknown )
{
    /*** IUnknown methods ***/
    STDMETHOD(QueryInterface) (THIS_ REFIID riid, LPVOID FAR * ppvObj)
    PURE;
    STDMETHOD_(ULONG,AddRef) (THIS) PURE;
    STDMETHOD_(ULONG,Release) (THIS) PURE;
    /*** IDirectDrawSurface methods ***/
    STDMETHOD(AddAttachedSurface)(THIS_ LPDIRECTDRAWSURFACE) PURE;
    STDMETHOD(AddOverlayDirtyRect)(THIS_ LPRECT) PURE;
    STDMETHOD(Blt)(THIS_ LPRECT,LPDIRECTDRAWSURFACE, LPRECT,DWORD,
        LPDDBLTFX) PURE;
    STDMETHOD(BltBatch)(THIS_ LPDDBLTBATCH, DWORD, DWORD ) PURE;
    STDMETHOD(BltFast)(THIS_ DWORD,DWORD,LPDIRECTDRAWSURFACE,
        LPRECT,DWORD) PURE;
    STDMETHOD(DeleteAttachedSurface)(THIS_ DWORD,LPDIRECTDRAWSURFACE)
        PURE;
    STDMETHOD(EnumAttachedSurfaces)(THIS_ LPVOID,
        LPDDENUMSURFACESCALLBACK) PURE;
    STDMETHOD(EnumOverlayZOrders)(THIS_ DWORD, LPVOID,
        LPDDENUMSURFACESCALLBACK) PURE;
    STDMETHOD(Flip)(THIS_ LPDIRECTDRAWSURFACE, DWORD) PURE;
    STDMETHOD(GetAttachedSurface)(THIS_ LPDDSCAPS,
        LPDIRECTDRAWSURFACE FAR *) PURE;
    STDMETHOD(GetBltStatus)(THIS_ DWORD) PURE;
    STDMETHOD(GetCaps)(THIS_ LPDDSCAPS) PURE;
    STDMETHOD(GetClipper)(THIS_ LPDIRECTDRAWCLIPPER FAR*) PURE;
    STDMETHOD(GetColorKey)(THIS_ DWORD, LPDDCOLORKEY) PURE;
    STDMETHOD(GetDC)(THIS_ HDC FAR *) PURE;
    STDMETHOD(GetFlipStatus)(THIS_ DWORD) PURE;
    STDMETHOD(GetOverlayPosition)(THIS_ LPLONG, LPLONG ) PURE;
    STDMETHOD(GetPalette)(THIS_ LPDIRECTDRAWPALETTE FAR*) PURE;
    STDMETHOD(GetPixelFormat)(THIS_ LPDDPIXELFORMAT) PURE;
    STDMETHOD(GetSurfaceDesc)(THIS_ LPDDSURFCEDESC) PURE;
    STDMETHOD(Initialize)(THIS_ LPDIRECTDRAW, LPDDSURFCEDESC) PURE;
    STDMETHOD(IsLost)(THIS) PURE;
    STDMETHOD(Lock)(THIS_ LPRECT,LPDDSURFCEDESC,DWORD,HANDLE) PURE;
    STDMETHOD(ReleaseDC)(THIS_ HDC) PURE;
    STDMETHOD(Restore)(THIS) PURE;
    STDMETHOD(SetClipper)(THIS_ LPDIRECTDRAWCLIPPER) PURE;

```

```

STDMETHOD(SetColorKey)(THIS_ DWORD, LPDDCOLORKEY) PURE;
STDMETHOD(SetOverlayPosition)(THIS_ LONG, LONG ) PURE;
STDMETHOD(SetPalette)(THIS_ LPDIRECTDRAWPALETTE) PURE;
STDMETHOD(Unlock)(THIS_ LPVOID) PURE;
STDMETHOD(UpdateOverlay)(THIS_ LPRECT,
    LPDIRECTDRAWSURFACE, LPRECT, DWORD, LPDDOVERLAYFX) PURE;
STDMETHOD(UpdateOverlayDisplay)(THIS_ DWORD) PURE;
STDMETHOD(UpdateOverlayZOrder)(THIS_ DWORD, LPDIRECTDRAWSURFACE)
    PURE;
};

```

DirectDraw Return Values

DD_OK

All DirectDraw functions will return a value of **DD_OK** if the requested operation was successfully performed.

DirectDraw Enumeration Call Back Return Values

DDENUM_CANCEL

The requested enumeration has been stopped.

DDENUMRET_OK

The requested enumeration was successful or has been continued.

DirectDraw Error Return Values

DDERR_ALREADYINITIALIZED

This object already been initialized.

DDERR_BLTFASTCANTCLIP

A DirectDrawClipper object has been attached to the source surface in a **BlitFast** call.

DDERR_CANNOTATTACHSURFACE

The requested surface attachment cannot be performed.

DDERR_CANNOTDETACHSURFACE

The requested surface detachment cannot be performed.

DDERR_CANTCREATEDC

The device context could not be created. Windows can cause this error.

DDERR_CANTDUPLICATE

The requested surface could not be duplicated.

DDERR_CLIPPERISUSINGHWND

The DirectDrawClipper object already contains an HWND clip list.

DDERR_COLORKEYNOTSET

A source color key wasn't specified.

DDERR_CURRENTLYNOTAVAIL

Support is currently not available.

DDERR_DIRECTDRAWALREADYCREATED

A DirectDraw object representing this driver has already been created.

DDERR_EXCEPTION

An exception was encountered while performing this operation.

DDERR_EXCLUSIVEMODEALREADYSET

Exclusive mode has already been set.

DDERR_GENERIC

Generic failure.

DDERR_HEIGHTALIGN

RECT structure contains an invalid height value.

DDERR_HWNDALEADYSET

An attempt was made to change the cooperation level for a given HWND. DirectDrawSurface and DirectDrawPalette objects must be released before a change in cooperation level can take place.

DDERR_HWNDSubCLASSed

HWND used by **SetCooperativeLevel** has been subclassed, this prevents DirectDraw from restoring the previous state.

DDERR_IMPLICITLYCREATED

Implicitly surfaces cannot be restored.

DDERR_INCOMPATIBLEPRIMARY

Based on input criteria, cannot create the primary surface.

DDERR_INVALIDCAPS

One or more of the caps bits passed to the callback are incorrect.

DDERR_INVALIDCLIPLIST

The clip list cannot be attached to the DirectDrawClipper object.

DDERR_INVALIDDIRECTDRAWGUID

The GUID passed to **DirectDrawCreate** is not valid.

DDERR_INVALIDMODE

The requested video mode is not supported by DirectDraw.

DDERR_INVALIDOBJECT

A Pointer to a DirectDraw object is invalid.

DDERR_INVALIDPARAMS

A function input parameter is invalid.

DDERR_INVALIDPIXELFORMAT

The pixel format is invalid.

DDERR_INVALIDPOSITION

The position of the overlay on the destination is not a legal position.

DDERR_INVALIDRECT

RECT structure contains a bad value.

DDERR_LOCKEDSURFACES

More of the surfaces in the requested operation are locked.

DDERR_NO3D

No 3D hardware.

DDERR_NOALPHAHW

No alpha-capable hardware.

DDERR_NOANTITEARHW

No hardware support for synchronizing blts.

DDERR_NOBLTHW

No bltter.

DDERR_NOBLTQUEUEHW

No asynchronous blter.

DDERR_NOCLIPLIST

No clip list available.

DDERR_NOCLIPPERATTACHED

The surface doesn't have an attach DirectDrawClipper object.

DDERR_NOCOLORCONVHW

No color conversion hardware.

DDERR_NOCOLORKEY

The surface doesn't have a color key.

DDERR_NOCOLORKEYHW

The hardware can't perform destination color keying.

DDERR_NOCOOPERATIVELEVELSET

A cooperation level hasn't been established for the DirectDraw object.

DDERR_NODC

A DC hasn't been obtained for the surface.

DDERR_NODDROPSHW

No DirectDraw ROP hardware.

DDERR_NODIRECTDRAWHW

DDERR_NOEMULATION

Software emulation is not available.

DDERR_NOEXCLUSIVEMODE

The requested operation requires exclusive level.

DDERR_NOFLIPHW

The hardware cannot flip visible surfaces.

DDERR_NOGDI

The GDI is not present.

DDERR_NOHWND

The attempted operation requires an HWND parameter that has been previously set as the CooperativeLevel HWND.

DDERR_NOMIRRORHW

The hardware cannot perform the mirroring operation.

DDERR_NOOVERLAYDEST

Returned when **GetOverlayPosition** is called on an overlay that has never been called on to establish a destination. **UpdateOverlay**

DDERR_NOOVERLAYHW

The hardware does not support overlays.

DDERR_NOPALETTEATTACHED

A DirectDrawPalette object has not been attached to the surface.

DDERR_NOPALETTEHW

The hardware does not support palettes.

DDERR_NORASTEROPHW

The hardware does not support ROPS.

DDERR_NOROTATIONHW

The hardware does not support rotation.

DDERR_NOSTRETCHHW

The hardware does not support stretching.

DDERR_NOT4BITCOLOR

The requested operation requires a 4 bit palette.

DDERR_NOT4BITCOLORINDEX

The requested operation requires a 4 bit, double indexed palette.

DDERR_NOT8BITCOLOR

The requested operation requires an 8 bit display mode.

DDERR_NOTAOVERLAYSURFACE

The requested operation requires an overlay surface.

DDERR_NOTTEXTUREHW

The hardware does not support 3D textures.

DDERR_NOTFLIPPABLE

The surface does not allow flipping.

DDERR_NOTFOUND

Requested item was not found.

DDERR_NOTLOCKED

The surface is not currently locked.

DDERR_NOTPALETTIZED

The surface is not a palette surface.

DDERR_NOVSYNCHW

The hardware does not support VBI sync.

DDERR_NOZBUFFERHW

The hardware does not support Z buffers.

DDERR_NOZOVERLAYHW

The hardware cannot sort overlays based on a Z order.

DDERR_OUTOFCAPS

The requested hardware has already been allocated.

DDERR_OUTOFMEMORY

There is not enough system memory to perform the given operation.

DDERR_OUTOFVIDEOMEMORY

There is not enough video memory to perform the given operation.

DDERR_OVERLAYCANTCLIP

The hardware does not support clipped overlays.

DDERR_OVERLAYCOLORKEYONLYONEACTIVE

The hardware can only have one color key active.

DDERR_OVERLAYNOTVISIBLE

GetOverlayPosition was called for a hidden overlay.

DDERR_PALETTEBUSY

The palette is currently being accessed.

DDERR_PRIMARYSURFACEALREADYEXISTS

A primary surface has already been created.

DDERR_REGIONTOOSMALL

Region data passed to **GetClipList** is too small.

DDERR_SURFACEALREADYATTACHED

The surface is already attached.

DDERR_SURFACEALREADYDEPENDENT

This surface is already a dependency of the main surface.

DDERR_SURFACEBUSY

The surface is locked or is being accessed by another thread.

DDERR_SURFACEISOBSCURED

The surface is currently obscured.

DDERR_SURFACELOST

The surface memory associated with the DirectDrawSurface object has been deallocated.

DDERR_SURFACENOTATTACHED

The surface is not attached to another surface.

DDERR_TOOBIGHEIGHT

Requested height is too big.

DDERR_TOOBIGSIZE

Requested size by is too large—the individual height and width are OK.

DDERR_TOOBIGWIDTH

Requested width is too wide.

DDERR_UNSUPPORTED

Action is not supported.

DDERR_UNSUPPORTEDFORMAT

FOURCC code is not supported by DirectDraw.

DDERR_UNSUPPORTEDMASK

Invalid bitmask value.

DDERR_VERTICALBLANKINPROGRESS

A vertical blank interval is in progress.

DDERR_WASSTILLDRAWING

The requested action was terminated because a previous activity was still taking place on the surface.

DDERR_WRONGMODE

The surface could not be restored, it was created in a different display mode.

DDERR_XALIGN

RECT structure was not horizontally aligned on required boundary.

ModeX Surfaces

DirectDraw allows the ModeX display modes of 320×240 and 320×200 . Unlike a traditional ModeX implementation, where the game application assumes the entire responsibility for the ModeX display and its graphics organization, DirectDraw's version of ModeX provides a level of abstraction from the details of the ModeX display. While this approach has its advantages, it doesn't come without trade offs—the most significant being a decrease in performance over a low-level ModeX system.

To display a ModeX video mode, the DirectDraw program must include the **DDSCL_ALLOWMODEX** flag in the **SetCooperativeLevel** function. A ModeX display also requires the **DDSCL_EXCLUSIVE** and **DDSCL_FULLSCREEN** flags. The application can then set a ModeX display by calling the **SetDisplayMode** function with either (320, 200, 8) or (320, 240, 8) as display setting parameters.

MODEX RESTRICTIONS

ModeX programs cannot lock, blit, or get a device context to the primary surface. The ModeX program must blit its graphics to a back buffer surface, and then call the **Flip** function for the new frame of graphics to become visible.

DirectDraw implements the ModeX display by creating two planar (and hidden) ModeX surfaces in the VGA card's memory. One of these ModeX surfaces will be visible to the user at any given time (the primary ModeX surface). In ModeX, the **Flip** function actually instigates a copy of a linear back buffer to one of the planar ModeX surfaces. The internal ModeX surfaces are then page flipped by DirectDraw and a new frame of graphics is seen. The translation from linear surfaces to planar ModeX surfaces is the main reason why a DirectDraw ModeX program isn't as fast as a legitimate ModeX program, where a linear to planar translation is usually avoided.

When programming in DirectDraw's version of ModeX, all surfaces, except the primary surface are linear and fully accessible by the DirectDraw application. The primary surface, maintained by DirectDraw, represents the data in a back buffer after it has been translated to planar ModeX.

THE MODEX.CPP PROGRAM

The following program listing is the CLIP.CPP program from chapter 8 modified to display either ModeX or an SVGA 640 × 480 video mode. Within the program, the **Spacebar** toggles between ModeX and SVGA displays.

```

//*****
//DirectDraw Programming
//Bret M. Timmins
//
//(C)1995 M&T Books
//
//MODEX.cpp
// Displays a graphic in 640x480 fullscreen or ModeX
//*****
#define WIN32_LEAN_AND_MEAN
#include <windows.h>
#include <windowsx.h>
#if defined( __BORLANDC__ ) && defined( __WIN32__ )
#define _WIN32
#endif
#include <ddraw.h>
#include <stdio.h>
#include <stdlib.h>
#include <stdarg.h>

LPDIRECTDRAW          lpDirectDrawObject; //DD object
LPDIRECTDRAWSURFACE lpPrimary;           //DD primary surface
LPDIRECTDRAWSURFACE lpBackbuffer;        //DD back buffer surface
LPDIRECTDRAWSURFACE lpOffscreen;         //Offscreen image
LPDIRECTDRAWSURFACE lpOffbuffer;         //Surface buffer for the
                                         //entire display surface

LPDIRECTDRAWCLIPPER lpClipper;           //Clipper object
BOOL                 ActiveApp;          //Is this program active?

int PASCAL WinMain( HINSTANCE hInstance, HINSTANCE hPrevInstance,
                    LPSTR lpCmdLine, int nCmdShow);
long FAR PASCAL WindowProc ( HWND hwnd, UINT message,
                             WPARAM wParam, LPARAM lParam );
BOOL InitDDSurfaces ( HWND hwnd );

```

```

BOOL DrawFrame ( void );
BOOL RestoreSurfaces ( void );
BOOL BMPToDirectDrawSurface ( LPDIRECTDRAWSURFACE surface,
                               char *bmp );
LPDIRECTDRAWPALETTE GetBMPDDPalette(LPDIRECTDRAWSURFACE surface,
                                     char *bmp );

int ScreenWidth = 320;
int ScreenHeight = 240;
int SurfaceWidth = 516/2;
int SurfaceHeight = 204/2;

int XPos,YPos;

HWND Globalhwnd;

//*****
//WinMain - mandatory windows init function
//
//*****
int PASCAL WinMain( HINSTANCE hInstance, HINSTANCE hPrevInstance,
                   LPSTR lpCmdLine, int nCmdShow)
{
    MSG         msg;
    HWND        hwnd;
    WNDCLASS    wc;
    static char ClassName[] = "Modex";
    HRESULT     ddreturn;

    lpCmdLine = lpCmdLine;
    hPrevInstance = hPrevInstance;

    //register and realize our display window
    wc.style = CS_HREDRAW | CS_VREDRAW;
    wc.lpfnWndProc = WindowProc;
    wc.cbClsExtra = 0;
    wc.cbWndExtra = 0;
    wc.hInstance = hInstance;
    wc.hIcon = LoadIcon( hInstance, IDI_APPLICATION );
    wc.hCursor = LoadCursor( NULL, IDC_ARROW );
    wc.hbrBackground = NULL;
    wc.lpszMenuName = ClassName;
    wc.lpszClassName = ClassName;
    RegisterClass( &wc );

    //Create a full screen window so that GDI won't ever be
    //called
    hwnd = CreateWindowEx(
        WS_EX_TOPMOST,
        ClassName,

```

```

ClassName,
WS_POPUP,
0,
0,
GetSystemMetrics(SM_CXSCREEN),
GetSystemMetrics(SM_CYSCREEN),
NULL,
NULL,
hInstance,
NULL );

if( !hwnd )
    return FALSE;

ShowWindow( hwnd, nCmdShow );
UpdateWindow( hwnd );
SetFocus( hwnd );
ShowCursor( FALSE );

//Instantiate our DirectDraw object
ddreturn = DirectDrawCreate( NULL,
                             &lpDirectDrawObject, NULL );
if ( ddreturn != DD_OK )
{
    DestroyWindow( hwnd );
    return FALSE;
}

ddreturn = lpDirectDrawObject->SetCooperativeLevel( hwnd,
                                                    DDSCL_EXCLUSIVE | DDSCL_FULLSCREEN |
                                                    DDSCL_ALLOWMODEX );
if ( ddreturn != DD_OK )
{
    DestroyWindow( hwnd );
    return FALSE;
}

if ( !InitDDSurfaces ( hwnd ) )
{
    DestroyWindow( hwnd );
    return FALSE;
}

if ( ScreenWidth == 320 )
    XPos = 50;
else
    XPos = 100;
YPos = SurfaceHeight * (-1);

Globalhwnd = hwnd;

```



```

while( 1 )
{
    if( PeekMessage( &msg, NULL, 0, 0, PM_NOREMOVE ) )
    {
        if( !GetMessage( &msg, NULL, 0, 0 ) )
            return msg.wParam;
        TranslateMessage(&msg);
        DispatchMessage(&msg);
    }else if ( ActiveApp )
    {
        DrawFrame ();
    }else
    {
        WaitMessage ();
    }
}

} //WinMain

//*****
// InitDDSurfaces - create the directdraw surfaces at
// either 640 x 480 or 320 x 240 Modex. If modex, use blt
// to scale the graphics
//
//*****
BOOL InitDDSurfaces ( HWND hwnd )
{
    DDSURFCEDESC    ddsd;
    DDSCAPS          ddscaps;
    DDCOLORKEY       ddck;
    HRESULT           ddreturn;
    LPDIRECTDRAWSURFACE lpTemporary;

    //Set the video mode to (globals) ScreenWidth and
    //ScreenHeight
    ddreturn = lpDirectDrawObject->SetDisplayMode( ScreenWidth,
                                                    ScreenHeight, 8);

    if ( ddreturn != DD_OK )
        return FALSE;

    // Create the primary surface and one back buffer surface
    ddsd.dwSize = sizeof( ddsd );
    ddsd.dwFlags = DDSD_CAPS | DDSD_BACKBUFFERCOUNT;
    ddsd.ddsCaps.dwCaps = DDSCAPS_PRIMARYSURFACE |
                        DDSCAPS_FLIP |
                        DDSCAPS_COMPLEX;
    ddsd.dwBackBufferCount = 1;
    ddreturn = lpDirectDrawObject->CreateSurface( &ddsd,
                                                    &lpPrimary, NULL );

    if ( ddreturn != DD_OK )

```

Appendix B: ModeX Surfaces

```
        return FALSE;

//get a surface pointer to our back buffer
ddscaps.dwCaps = DDSCAPS_BACKBUFFER;
ddreturn = lpPrimary->GetAttachedSurface (&ddscaps,
                                           &lpBackbuffer);

if( ddreturn != DD_OK )
    return FALSE;

// Create a clipper
ddreturn = lpDirectDrawObject->CreateClipper ( 0, &lpClipper,
                                                NULL );

if ( ddreturn != DD_OK )
    return FALSE;

// Get coordinates of our window
ddreturn = lpClipper->SetHwnd ( 0, hwnd );
if ( ddreturn != DD_OK )
    return FALSE;

// Apply Clipper to the back buffer
ddreturn = lpBackbuffer->SetClipper( lpClipper );
if ( ddreturn != DD_OK )
    return FALSE;

// Create a buffer for display memory
ddsd.dwFlags = DDSD_CAPS | DDSD_HEIGHT | DDSD_WIDTH;
ddsd.ddsCaps.dwCaps = DDSCAPS_OFFSCREENPLAIN;
ddsd.dwHeight = ScreenHeight;
ddsd.dwWidth = ScreenWidth;
ddreturn = lpDirectDrawObject->CreateSurface ( &ddsd,
                                              &lpOffbuffer,
                                              NULL );

if ( ddreturn != DD_OK )
    return FALSE;

// Create the main offscreen surface
ddsd.dwFlags = DDSD_CAPS | DDSD_HEIGHT | DDSD_WIDTH;
ddsd.ddsCaps.dwCaps = DDSCAPS_OFFSCREENPLAIN;
ddsd.dwHeight = SurfaceHeight;
ddsd.dwWidth = SurfaceWidth;
ddreturn = lpDirectDrawObject->CreateSurface ( &ddsd,
                                              &lpOffscreen,
                                              NULL );

if ( ddreturn != DD_OK )
    return FALSE;

//Set the transparent color key for the off screen surface
ddck.dwColorSpaceLowValue = 255;
ddck.dwColorSpaceHighValue = 255;
```

```

lpOffscreen->SetColorKey( DDCKEY_SRCBLT, &ddck );

//At this point, we have the surfaces that the program
//will require. But, the graphic's for the program are sized
//for a 640x480 display. If we are running in Modex, we
//must scale them.
if ( ScreenWidth == 320 )
{
    // Create the temporary surface
    ddsd.dwFlags = DDSD_CAPS | DDSD_HEIGHT | DDSD_WIDTH;
    ddsd.ddsCaps.dwCaps = DDSCAPS_OFFSCREENPLAIN |
        DDSCAPS_SYSTEMMEMORY;
    ddsd.dwHeight = ScreenHeight*2;
    ddsd.dwWidth = ScreenWidth*2;
    ddreturn = lpDirectDrawObject->CreateSurface ( &ddsd,
                                                    &lpTemporary,
                                                    NULL );

    if ( ddreturn != DD_OK )
        return FALSE;

    //copy the background BMP to the temporary surface
    if (!BMPToDirectDrawSurface(lpTemporary,"desert.bmp" ))
        return FALSE;

    //Use blt to scale the graphic
    RECT Src = {0,0,640,480};
    RECT Dest = {0,0,320,240};
    ddreturn = lpBackbuffer->Blt( &Dest,
                                   lpTemporary, &Src,
                                   DDBLT_WAIT,NULL );

    if ( ddreturn != DD_OK )
        return FALSE;

    //Fill the Offbuffer
    ddreturn = lpOffbuffer->Blt( &Dest,
                                   lpTemporary, &Src,
                                   DDBLT_WAIT,NULL );

    if ( ddreturn != DD_OK )
        return FALSE;

    lpTemporary->Release();

    // Create the temporary surface
    ddsd.dwSize = sizeof( ddsd );
    ddsd.dwFlags = DDSD_CAPS | DDSD_HEIGHT | DDSD_WIDTH;
    ddsd.ddsCaps.dwCaps = DDSCAPS_OFFSCREENPLAIN |
        DDSCAPS_SYSTEMMEMORY;
    ddsd.dwHeight = SurfaceHeight*2;
    ddsd.dwWidth = SurfaceWidth*2;
    ddreturn = lpDirectDrawObject->CreateSurface ( &ddsd,

```

```

                                                                    &lpTemporary,
                                                                    NULL );

    if ( ddreturn != DD_OK )
        return FALSE;

    //scale the foreground graphic
    Src.left = 0; Src.top = 0;
    Src.right = 516; Src.bottom = 204;
    Dest.top = 0; Dest.top = 0;
    Dest.right = 516/2; Dest.bottom = 204/2;

    //copy the background BMP to the temporary surface
    if (!BMPToDirectDrawSurface(lpTemporary,"arizona.bmp"))
        return FALSE;

    //Use blt to scale the graphic
    ddreturn = lpOffscreen->Blt( &Dest,
                                lpTemporary, &Src,
                                DDBLT_WAIT,NULL );
    if ( ddreturn != DD_OK )
        return FALSE;

    lpTemporary->Release();

    //Palette Case: In modex, we can't write to the primary
    //surface. Therefore, we can't easily attach a palette
    //to the primary surface using BMPToDirectDrawSurface
    //For modeX, we will have to manually attach a palette
    //to the primary surface
    GetBMPDDPalette ( lpPrimary, "desert.bmp" );

}
else
{
    //copy a BMP file to the primary surface
    if (!BMPToDirectDrawSurface (lpPrimary,"desert.bmp"))
        return FALSE;

    //copy to the back buffer
    if (!BMPToDirectDrawSurface(lpBackbuffer,"desert.bmp"))
        return FALSE;

    //copy to the display buffer
    if ( !BMPToDirectDrawSurface (lpOffbuffer,"desert.bmp"))
        return FALSE;

    //copy to the offscreen surface
    if (!BMPToDirectDrawSurface (lpOffscreen,"arizona.bmp"))
        return FALSE;
}
return TRUE;

```

```
//InitDDSurfaces

//*****
//WindowProc - receive and handle windows messages
//
//*****
long FAR PASCAL WindowProc( HWND hwnd, UINT message,
                           WPARAM wParam, LPARAM lParam )
{
    switch( message )
    {
        case WM_ACTIVATEAPP:
            ActiveApp = wParam;
            break;

        case WM_CREATE:
            break;

        case WM_KEYDOWN:
            switch ( wParam )
            {
                case VK_ESCAPE:
                    DestroyWindow ( hwnd );
                    break;

                case VK_SPACE:
                    if (ScreenWidth == 640 )
                    {
                        ScreenWidth = 320;
                        ScreenHeight = 240;
                        SurfaceWidth = 516/2;
                        SurfaceHeight = 204/2;
                    }else
                    {
                        ScreenWidth = 640;
                        ScreenHeight = 480;
                        SurfaceWidth = 516;
                        SurfaceHeight = 204;
                    }
                    if ( ScreenWidth == 320 )
                        XPos = 50;
                    else
                        XPos = 100;
                    YPos = SurfaceHeight * (-1);

                    //Force the clipper to update its size
                    MoveWindow (hwnd,0,0,ScreenWidth,
                               ScreenHeight,FALSE);

                    lpOffscreen->Release();
            }
    }
}
```

```

        lpOffbuffer->Release();
        lpPrimary->Release();
        lpClipper->Release();

        if ( !InitDDSurfaces ( hwnd ) )
        {
            DestroyWindow ( hwnd );
            break;
        }

        DrawFrame ();
        break;
    }
    break;

case WM_DESTROY:
    if ( lpDirectDrawObject != NULL )
    {
        if ( lpOffscreen != NULL )
            lpOffscreen->Release();
        if ( lpOffbuffer != NULL )
            lpOffbuffer->Release();
        if ( lpPrimary != NULL )
            lpPrimary->Release();
        lpDirectDrawObject->Release();
    }

    ShowCursor ( TRUE );
    PostQuitMessage ( wParam );
    break;

default:
    return DefWindowProc(hwnd,message,wParam,lParam);
}

return 0L;

} //WindowProc

//*****
//DrawFrame - Compose a frame of graphics, flip surfaces
//
//return:
//  BOOL TRUE - success
//*****
BOOL DrawFrame ( void )
{
    HRESULT ddreturn;
    RECT ImageRect;
    RECT BufferRect;

```

```

RECT AnimRect;

ImageRect.left = 0; ImageRect.top = 0;
ImageRect.right = SurfaceWidth;
ImageRect.bottom = SurfaceHeight;
BufferRect.left = 0; BufferRect.top = 0;
BufferRect.right = ScreenWidth;
BufferRect.bottom = ScreenHeight;

//clean up the back buffer from the previous frame by blting
//the pixel data on the display buffer to the back buffer
do
{
    ddreturn = lpBackbuffer->Blit( &BufferRect,
                                   lpOffbuffer, &BufferRect,
                                   DDBLT_WAIT, NULL );
    if ( ddreturn == DDERR_SURFACELOST )
    {
        if ( !RestoreSurfaces() )
            return FALSE;
    }
}while ( ddreturn != DD_OK );

//Blit the offscreen surface to the back buffer
AnimRect.left = XPos;
AnimRect.top = YPos;
AnimRect.right = XPos+SurfaceWidth;
AnimRect.bottom = YPos+SurfaceHeight;
do
{
    ddreturn = lpBackbuffer->Blit( &AnimRect,
                                   lpOffscreen, &ImageRect, DDBLT_KEYSRC|
                                   DDBLT_WAIT, NULL );
    if ( ddreturn == DDERR_SURFACELOST )
    {
        if ( !RestoreSurfaces() )
            return FALSE;
    }
}while ( ddreturn != DD_OK );

YPos++;
if ( YPos > ScreenHeight )
    YPos = SurfaceHeight * (-1);

//Let DirectDraw switch our surface pointers
do
{
    ddreturn = lpPrimary->Flip( NULL, DDFLIP_WAIT );
    if ( ddreturn == DDERR_SURFACELOST )
    {

```

Appendix B: ModeX Surfaces

```

        if ( !RestoreSurfaces() )
            return FALSE;
    }
}while ( ddreturn != DD_OK );

return TRUE;

} // DrawFrame

// *****
// RestoreSurfaces - restores all lost surfaces
//
// returns:
//  BOOL True - success
// *****
BOOL RestoreSurfaces ( void )
{
    MoveWindow (Globalhwnd,0,0,ScreenWidth,ScreenHeight,FALSE);

    lpOffscreen->Release();
    lpOffbuffer->Release();
    lpPrimary->Release();
    lpClipper->Release();

    if ( !InitDDSurfaces ( Globalhwnd ) )
        return FALSE;

    return TRUE;
} // RestoreSurfaces

// *****
// BMPToDirectDrawSurface - opens and copies a BMP to a DD surface
// input:
//  LPDIRECTDRAWSURFACE surface
//  char *bmp - filename
// returns:
//  BOOL TRUE - success
// *****
BOOL BMPToDirectDrawSurface ( LPDIRECTDRAWSURFACE surface,
                               char *bmp )
{
    HRESULT          ddreturn;
    DWORD            actualRead;
    HANDLE           hfile;
    char             path[MAX_PATH];
    BITMAPFILEHEADER BMPFileHead;
    BITMAPINFOHEADER BMPFileInfo;
    RGBQUAD          Palette[256];
    DDSURFACEDESC    ddsd;

```



```
int          i,src_width;
BYTE         *lpDDMemory,*lpBMPMemory,*image;

hfile = CreateFile( bmp, GENERIC_READ, FILE_SHARE_READ,
                    (LPSECURITY_ATTRIBUTES)NULL,
                    OPEN_EXISTING, FILE_ATTRIBUTE_NORMAL,
                    (HANDLE) NULL );

if ( hfile == INVALID_HANDLE_VALUE )
{
    strcpy (path,"..\\");           //back up one directory
    strcat( path, bmp );
    hfile = CreateFile( path, GENERIC_READ, FILE_SHARE_READ,
                        (LPSECURITY_ATTRIBUTES)NULL,
                        OPEN_EXISTING, FILE_ATTRIBUTE_NORMAL,
                        (HANDLE) NULL );
    if ( hfile == INVALID_HANDLE_VALUE )
        return FALSE;
}

// Read the BMP header and info structures
if( !ReadFile ( hfile, &BMPFileHead, sizeof ( BMPFileHead ),
               &actualRead, NULL) )
    return FALSE;

if( !ReadFile ( hfile, &BMPFileInfo, sizeof ( BMPFileInfo ),
               &actualRead, NULL) )
    return FALSE;

//Make sure the BMP is 8-bit color
if ( BMPFileInfo.biBitCount != 8 )
    return FALSE;

//get the BMP palette
if ( !ReadFile ( hfile, Palette, sizeof( Palette ),
               &actualRead, NULL) )
    return FALSE;

//Allocate memory for image data
image = (BYTE *) LocalAlloc ( LPTR,
                             BMPFileInfo.biWidth *
                             BMPFileInfo.biHeight );

if( image == NULL )
    return FALSE;

//read BMP into memory
if( !ReadFile ( hfile, image,
               BMPFileInfo.biWidth * BMPFileInfo.biHeight,
               &actualRead, NULL) )
{
```

```

        LocalFree( image );
        return FALSE;
    }

    //copy BMP data to DirectDraw surface
    memset ( &ddsd, 0, sizeof( DDSURFACEDESC ) );
    ddsd.dwSize = sizeof( ddsd );
    ddreturn = surface->Lock( NULL, &ddsd, 0, NULL );

    if( ddreturn != DD_OK )
    {
        LocalFree( image );
        return FALSE;
    }

    lpDDMemory = (BYTE *) ddsd.lpSurface;
    lpBMPMemory = image+((BMPFileInfo.biHeight-1)*
                        BMPFileInfo.biWidth);

    if ( BMPFileInfo.biWidth > ddsd.lPitch )
        src_width = ddsd.lPitch;
    else
        src_width = BMPFileInfo.biWidth;

    for( i = 0; i < BMPFileInfo.biHeight; i++ )
    {
        memcpy ( lpDDMemory, lpBMPMemory, src_width );
        lpDDMemory += ddsd.lPitch;
        lpBMPMemory -= src_width;
    }

    surface->Unlock ( NULL );
    LocalFree( image );
    CloseHandle(hfile);

    //Get and set the palette
    if ( GetBMPDDPalette ( surface, bmp ) == NULL)
        return FALSE;

    return TRUE;
} //BMPToDirectDrawSurface

/*****
//GetBMPDDPalette - Gets a palette from a 8 bit BMP file
//Applies the palette to a DDSurface
//input:
// LPDIRECTDRAW lpDirectDrawObject - DirectDraw Object
// char *bmp - filename
//returns:

```

```
// A LPDIRECTDRAWPALETTE pointer
//*****
LPDIRECTDRAWPALETTE GetBMPDDPalette(LPDIRECTDRAWSURFACE surface,
                                     char *bmp )
{
    DWORD          actualRead;
    HANDLE          hfile;
    char            path[MAX_PATH];
    BITMAPFILEHEADER BMPFileHead;
    BITMAPINFOHEADER BMPFileInfo;
    RGBQUAD         Palette[256];
    PALETTEENTRY     pe[256];
    LPDIRECTDRAWPALETTE lpDDPalette;
    HRESULT          ddreturn;
    int i;

    hfile = CreateFile( bmp, GENERIC_READ, FILE_SHARE_READ,
                       (LPSECURITY_ATTRIBUTES)NULL,
                       OPEN_EXISTING, FILE_ATTRIBUTE_NORMAL,
                       (HANDLE) NULL );

    if ( hfile == INVALID_HANDLE_VALUE )
    {
        strcpy (path, "..\\");          //back up one directory
        strcat( path, bmp );
        hfile = CreateFile( path, GENERIC_READ, FILE_SHARE_READ,
                           (LPSECURITY_ATTRIBUTES)NULL,
                           OPEN_EXISTING, FILE_ATTRIBUTE_NORMAL,
                           (HANDLE) NULL );
        if ( hfile == INVALID_HANDLE_VALUE )
            return NULL;
    }

    // Read the BMP header and info structures
    if( !ReadFile ( hfile, &BMPFileHead, sizeof ( BMPFileHead ),
                   &actualRead, NULL) )
        return NULL;

    if( !ReadFile ( hfile, &BMPFileInfo, sizeof ( BMPFileInfo ),
                   &actualRead, NULL) )
        return NULL;

    //Make sure the BMP is 8-bit color
    if ( BMPFileInfo.biBitCount != 8 )
        return NULL;;

    //get the BMP palette
    if ( !ReadFile ( hfile, Palette, sizeof( Palette ),
                   &actualRead, NULL) )
        return NULL;
```

```
//create a DirectDraw palette for the surface
for( i=0; i<256; i++ )
{
    pe[i].peRed = Palette[i].rgbRed;
    pe[i].peGreen = Palette[i].rgbGreen;
    pe[i].peBlue = Palette[i].rgbBlue;
}
ddreturn = lpDirectDrawObject->CreatePalette( DDPCAPS_8BIT,
                                              pe,
                                              &lpDDPalette,
                                              NULL );

if ( ddreturn != DD_OK )
    return FALSE;
surface->SetPalette ( lpDDPalette );

return lpDDPalette;
} //GetBMPDDPalette
```

Bibliography

- Abrash, Michael. *Zen of Graphics Programming*. Coriolis Group Books, 1994.
- Custer, Helen. *Inside Windows NT*. Microsoft Press, 1993.
- Ferraro, Richard E. *Programmer's Guide to the EGA and VGA Cards, 2nd Edition*. Addison-Wesley, 1990.
- Kernighan, Brian W., Dennis M. Ritchie. *The C Programming Language, 2nd Edition*. Prentice Hall, 1988.
- LaMothe Andre, John Ratcliff, Mark Seminatore and Denise Tyler. *Tricks of the Game Programming Gurus*. Sams Publishing, 1994.
- Microsoft Corporation. *Microsoft Win32 Programmer's Reference, Volume One*. Microsoft Press.
- Petzold, Charles. *Programming Windows*. Microsoft Press, 1990.
- Rimmer, Steve. *Bit-Mapped Graphics*. WindCrest, 1990.
- Stevens, Al, Stan Trujillo. *C++ Games Programming*. M&T Books, 1995.
- Stevens, Al. *Teach yourself C++*. MIS:Press, 1995.

4. The fourth part of the document discusses the role of the accounting system in providing financial information to management and the importance of using this information to make informed decisions. It emphasizes the need for accurate and timely financial information and the role of the accounting system in providing this information.

5. The fifth part of the document discusses the role of the accounting system in providing financial information to external stakeholders and the importance of using this information to build trust and confidence. It emphasizes the need for transparency and accountability in financial reporting and the role of the accounting system in providing this information.

6. The sixth part of the document discusses the role of the accounting system in providing financial information to the public and the importance of using this information to make informed decisions. It emphasizes the need for accurate and timely financial information and the role of the accounting system in providing this information.

7. The seventh part of the document discusses the role of the accounting system in providing financial information to the government and the importance of using this information to make informed decisions. It emphasizes the need for accurate and timely financial information and the role of the accounting system in providing this information.

8. The eighth part of the document discusses the role of the accounting system in providing financial information to the media and the importance of using this information to make informed decisions. It emphasizes the need for accurate and timely financial information and the role of the accounting system in providing this information.

9. The ninth part of the document discusses the role of the accounting system in providing financial information to the public and the importance of using this information to make informed decisions. It emphasizes the need for accurate and timely financial information and the role of the accounting system in providing this information.

10. The tenth part of the document discusses the role of the accounting system in providing financial information to the public and the importance of using this information to make informed decisions. It emphasizes the need for accurate and timely financial information and the role of the accounting system in providing this information.

I N D E X

A

AddAttachedSurface function, 253

AddOverlayDirtyRect function, 253-254

AddRef function

 DirectDraw::AddRef, 221

 DirectDrawClipper::AddRef, 241

 DirectDrawSurface::AddRef, 254

 DirectDrawPalette::AddRef, 247

Alpha blending, 5, 120

ANSI C compatibility, 11, 15

B

BACKBUFF.CPP example program, 50-58

Blting

 Blt function, 62, 117-120, 121, 140, 153, 255-259

 Blt special effects, 118-121

 BltBatch function, 259-261

 BltFast function, 62-63, 73, 74, 86, 87, 117-118,
 140, 153, 178, 261-262

 Blting hardware, 5, 20, 62, 73, 117, 120, 121,

Transparent blting, 62, 63, 87-88, 102-103, 119
BMP file format, 42-43, 172, 200
Borland C/C++, 10

C

CGA (Color Graphics Adapter), 2
CLIP.CPP example program, 140-151
Clip lists, 5, 135, 136, 140
Color fill, 118, 120, 174,
Color key, 87, 88-89, 102-103, 119, 174
Color space, 89
COM (Component Object Module), 14-15
Compact function, 221-222
CONNECT.CPP example program, 20-25
CreateClipper function, 135-136, 222
CreateDIBSection, 4,
CreatePalette function, 154, 155-156, 157, 173, 1
80, 223-224
CreateSurface function, 30, 31, 33, 42, 48, 49, 58,
60, 159, 224-225
CreateWindow, 25,
CreateWindowEx, 42, 136,

D

DDBLTBATCH structure, 292-294
DDBLTFX structure, 118, 119, 120, 121, 294-298

- DD_OK (*See also* HRESULT), 17
- DDCAPS structure, 19-20, 298-309
- DDCOLORKEY structure, 89, 90, 91, 309
- DDMODEDESC structure, 309-310
- DDOVERLAYFX structure, 310-312
- DDPIXELFORMAT structure, 159, 160, 312-315
- DDSCAPS structure, 31, 315-316
- DDSURFACEDESC structure, 30, 31, 33, 42, 48-49, 58, 60, 159, 316-319
- DEBUG.CPP example program, 204-217
- Debugging, 61
 - GDI based debugging, 201-203
 - Kernel level debugging, 203
- DeleteAttachedSurfaces function, 263
- Destination keying (*See also* Color key), 88, 103, 119
- Device independent programming, 13-14
- Drawing Surfaces
 - Attached surfaces, 46,
 - Back buffer surfaces 45, 47-48, 49, 58, 63, 73, 333
 - Basic concepts, 27-29
 - ModeX surfaces (*See also* ModeX), 333-334
 - Off screen surfaces, 59-61, 73, 86, 103, 172, 202
 - Primary surfaces, 27, 29-30, 31, 33, 42, 45, 47-48, 49, 58, 73, 172, 175, 176, 179, 199, 201-203, 333
 - Surface properties, 29, 159
 - Surface pitch, 29, 34, 60
 - Surface creation (*See also* CreateSurface), 30, 48-49, 60
 - System memory surfaces, 61, 202
 - Video memory surfaces, 27-28, 201, 202
- DirectDraw Clippers, 62

- Attaching to drawing surfaces (*See also* SetClipper), 139
- Clipper objects (*See also* CreateClipper), 135-136, 137, 139, 140, 176, 199
- Clipper theory, 133-135,
- Clipping windows (*See also* SetHwnd), 136
- Creating clip lists (*See also* Clip lists) 137-139
- DirectDraw COM, 14-15
- DirectDrawCreate function, 17, 179, 219-220
- DirectDrawEnumerate function, 220
- DirectDraw objects
 - Object creation (*See also* DirectDrawCreate), 17-19
 - Object cooperation level (*See also* SetCooperativeLevel), 18-19, 25, 30, 42, 176, 333
 - Object capabilities (*See also* GetCaps), 19
- DirectDraw palettes
 - Attaching to drawing surfaces (*See also* SetPalette), 156
 - Palette objects (*See also* CreatePalette), 154, 155-156
- DirectDraw return values
 - DirectDraw enumeration callback values, 322
 - DirectDraw error return values 322-332
- DirectDraw software emulation, 6, 15-16, 19, 62, 117
- DirectDraw software interface, 15-16, 17
- DIRECTDRAW structure, 319
- DIRECTDRAWCLIPPER structure, 320
- DIRECTDRAWPALETTE structure, 320
- DIRECTDRAWSURFACE structure, 321-322
- DirectDrawSurface object, 33-34, 47
- DirectDraw HAL (Hardware Abstraction Layer), 15-16
- DirectPlay, 7
- DirectSound, 2, 7

DirectX, 7
Display buffers, *See* Drawing surfaces,
 Primary surfaces, back buffer surfaces
Display buffer ring, 46-48
Double buffering, *See* Page flipping
DuplicateSurface function, 225-226

E

EGA (Enhanced Graphics Adapter), 2
The EnumAttachedSurfaces function, 264
The EnumDisplayModes function, 32, 226-228
The EnumOverlayZOrders function, 265-266
The EnumSurfaces function, 228-229

F

Flip function, 50, 73, 266-267, 333
FlipToGDISurface function (*See also* GDI), 179, 202, 230
Frame buffer, 27, 29-30

G

GDI (Graphics Device Interface), 4, 11, 13, 15, 16, 1
 8, 25, 42, 154, 175
 Clipping, 178-179
 Cooperation levels, 176

DC's (Device Contexts), 178-179
GDI and primary surface access, 175-178
Palette control, 178-180, 200, 201-202, 203
GetAttachedSurface function, 49-50, 58, 73, 267-268
GetBltStatus function, 268
GetCaps function
 DirectDraw::GetCaps, 19-20, 25, 90, 230-231
 DirectDrawPalette::GetCaps, 247-248
 DirectDrawSurface::GetCaps, 269
GetClipper function, 269-270
GetClipList function, 241-242
GetColorKey function, 90-91, 270-271
GetDC function, 178-179, 201, 271
GetDisplayMode function, 231
GetEntries function, 157-158, 248-249
GetFlipStatus function, 272
GetFourCCCodes function, 232
GetGDISurface function, 232-233
GetHWND function, 242-243
GetMonitorFrequency function, 233-234
GetOverlayPosition function, 273
GetPalette function, 274
GetPixelFormat function, 274-275
GetScanLine function, 234
GetSurfaceDesc function, 159, 275-276
GetVerticalBlankStatus function, 234-235
GSDK sample files, 180, 199

H

Hicolor (16-bit RGB), 158-159, 173
HICOLOR.CPP example program, 160-172
HRESULT, 17

I

Identity palette, 180, 200
Initialize function,
 DirectDraw::Initialize, 235
 DirectDrawClipper::Initialize, 243
 DirectDrawPalette::Initialize, 249-250
 DirectDrawSurface::Initialize, 276
IsClipListChanged function, 243-244
IsLost function, 276-277

L

Linear memory access, 2, 4, 28
Linear to planer translation (*See also* ModeX), 333
Lock function (*See also* Unlock), 34, 43, 60, 178, 201, 277-278
Logical palettes, 180

M

Microsoft Game Software Development Kit

 Components, 7

 Installation, 7-8,

 Video Drivers, 9-10

Microsoft Visual C/C++, 10

Microsoft Windows, 4

Mode 13H, 2-3, 4, 46, 59

ModeX, 2-3, 18, 333-334

MODEX.CPP example program, 334-348

O

OFF_SCR.CPP example program, 63-72

OFF_SCR1.CPP example program, 74-85

Overlay surfaces, 5, 20, 90

P

Page flipping, 3, 4, 45-48, 50, 58, 175, 176, 201, 202, 333

PALETTEENTRY structure, 156, 157, 158

Palettes (*See also* RGB), 2, 88, 89, 121, 153-154, 179, 200

PRIMARY.CPP program, 35-42

Q

QueryInterface function,

 DirectDraw::QueryInterface, 236

 DirectDrawClipper::QueryInterface, 244-245

 DirectDrawPalette::QueryInterface, 250-251

 DirectDrawSurface::QueryInterface, 279

R

RECT structure, 34, 62, 63, 86, 118, 121, 138, 177

Release function

 DirectDraw::Release, 23, 236-237

 DirectDrawClipper::Release, 245

 DirectDrawPalette::Release, 251

 DirectDrawSurface::Release, 35, 280

ReleaseDC function, 178-179, 201, 280-281

Restore function, 281

RestoreDisplayMode function, 32

RGB (*See also* Hicolor, Palettes, True color), 5, 88, 153,
 158-159, 160, 173

RGNDATA structure, 137-139

RGNDATAHEADER structure, 139

ROP (Raster Operation), 119,

S

SetClipList function, 136, 137, 139, 245-246
 SetClipper function, 139, 282
 SetColorKey function, 89-90, 91, 282-283
 SetCooperativeLevel function, 18-19, 25, 30, 42, 176, 237-238, 333
 SetDisplayMode Function, 32, 33, 60, 238-239, 333
 SetEntries function, 157-158, 251-252
 SetHWND function, 136, 137, 139, 176, 199, 246-247
 SetOverlayPosition function, 283-284
 SetPalette function, 156, 173, 180, 284-285
 Screen Resolution (*See also* SetDisplayMode), 2, 5
 SHRINK.CPP example program, 121-132
 Soft-ICE Debugger, 203
 Source keying (*See also* Color key), 88, 102, 119, 174
 StretchBlt function, 118
 StretchDIBBlt function, 118
 SVGA (Super Video Graphics Adapter), 1, 3-4, 5, 27, 28, 153, 334
 System palette, 180

T

TB_DEST.CPP example program, 103-115
 TB_SRC.CPP example program, 91-102
 True Color (24-bit RGB), 158, 173

U

Unlock function (*See also* Lock), 34-35, 43, 201, 285

UpdateOverlay function, 286-289

UpdateOverlayDisplay function, 289-290

UpdateOverlayZOrder function, 290-291

V

Vertical blank interval, 2, 5, 46

VESA (Video Electronics Standards Association), 3-4

VGA (Video Graphics Adapter), 2-3, 4, 27, 28, 153

Video Cards

 Changing video modes (*See also* SetDisplayMode), 32, 176

 Determining capabilities (*See also* GetCaps), 19

 DirectDraw support, 9-10

Video Memory, 20, 27, 29, 61

Video Modes, 2, 3, 5, 176

W

WaitForVerticalBlank function, 240

Watcom C/C++, 10

Win16Mutex, 201-202

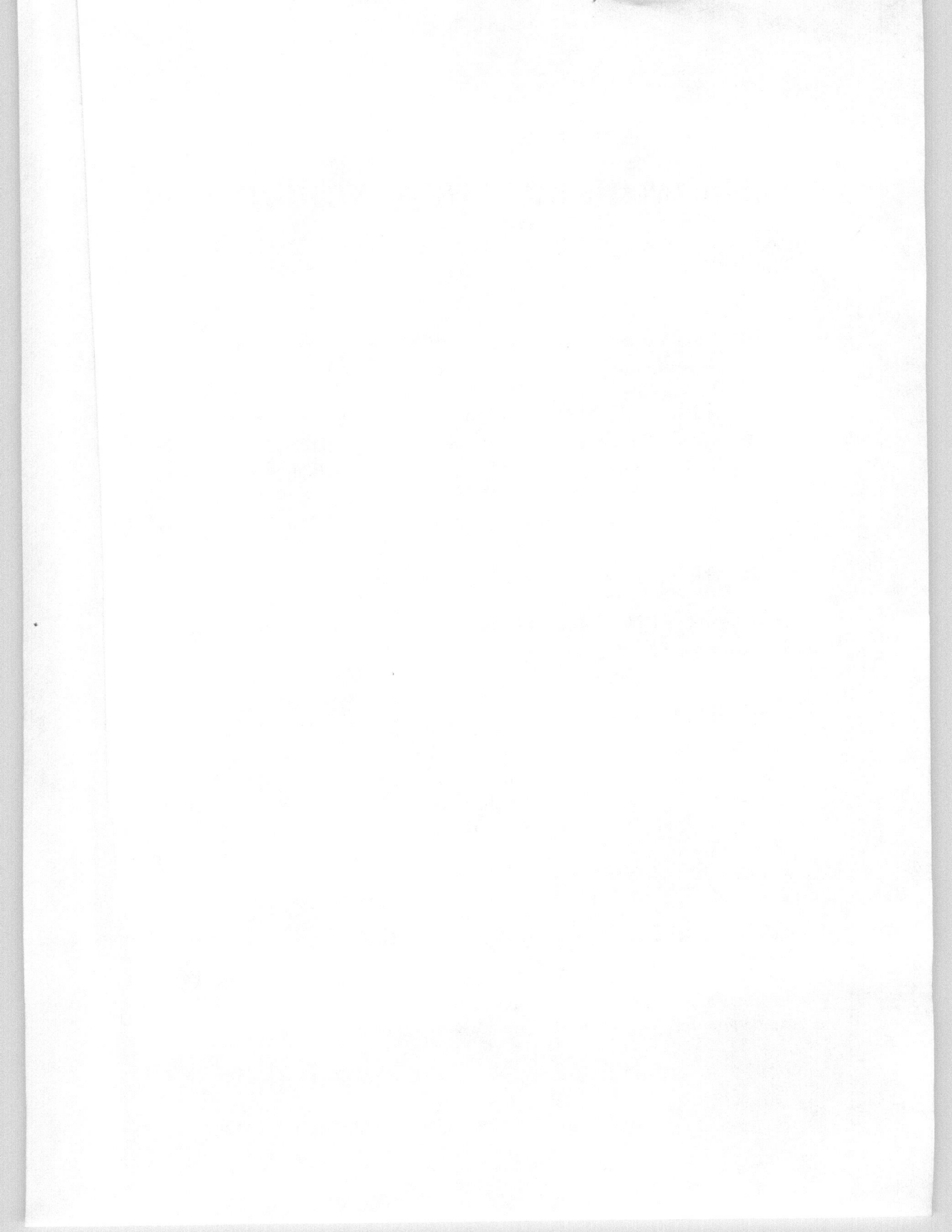
Win32 API, 11
Win32 SDK, 10
WINDBG Debugger, 203
WINDOW.CPP example program, 180-199
Windowed DirectDraw programming
 (See also GDI, FlipToGDISurface, SetHWND)
 Clipping, 176-177
 Cooperation levels, 176
 DC's (Device Contexts), 178-179
 GDI and primary surface access, 175-178
 Palette control, 179-180, 200
Windows Palette Manager, 179-180
Windows registry, 8
WinG, 4, 46

Y

YUV, 160

Z

Z buffers, 120



CD-ROM INSTALLATION INSTRUCTIONS

To install the C/C++ source code from the book, run SETUP.EXE from the root directory of the DirectDraw Programming companion CD-ROM. SETUP will copy the following items to your hard drive:

- The chapter examples with project files for Microsoft C/C++ 2.0 - 4.0, Borland C/C++ version 4.5 or higher, and Watcom C/C++ V10.5 or higher. Note that Watcom requires the Win32 SDK to compile DirectDraw programs. See the README.TXT file on the CD-ROM for additional information.
- Knob. A sample game written for DirectDraw and DirectSound featuring high speed parallax scrolling, animation, and multiple resolution support. The full source code for Knob is included.
- WTWIn. A DirectDraw port of Chris Laurel's 3D rendering demo, including source code.
- The DirectX runtime library.

The files that SETUP installs will require approximately 10 megabytes of hard disk space.

The Hive and Havoc

The CD-ROM includes playable demos of two cutting-edge games, The Hive™ from Trimark Interactive™ and Havoc from Reality Bytes™, which use DirectDraw and the rest of the Windows 95 Game SDK to enhance performance. To play The Hive, run the file THE_HIVE.EXE from \THEHIVE subdirectory. To play Havoc, run SETUP.EXE from the \HAVOC subdirectory. Both game demos can be copied to your hard drive for increased speed.

Additional Game Design Applications

The CD-ROM also includes shareware copies of ASL Painter! by ASL and WaveEdit by Keith W. Boone. Information on running ASL Painter! and WaveEdit can be found in the \ASLPAINT and \WAVEDIT subdirectories.

DirectDraw

PROGRAMMING

DirectDraw Programming is an eye-opening look at the single most important part of Microsoft's new Windows 95 Game SDK. DirectDraw finally allows Windows programmers to create the superfast graphics necessary for killer animated games and other graphic-intensive multimedia applications. Bret Timmins uses tutorials, an arsenal of sample code, and the step-by-step construction of an arcade-style game to show programmers how to get the most out of this radical new API for Windows 95. Learn the advantages of switching over from DOS or the now-obsolete WinG, and find out why the most intense new Windows-platform games are utilizing this leading-edge tool.

From selecting a video driver and using the right C/C++ compiler to installing the SDK and configuring your system, you'll find everything you need to begin programming in DirectDraw. You'll quickly master the terminology and fundamentals of objects, surfaces, and blitting. Then you'll learn to take advantage of the functional features of DirectDraw, including page flipping, access to blit hardware, multiple resolutions and color depths, special surface support, frame rate timing, clip lists, non-standard color models, and software emulation. Everything you need to begin programming commercial-quality games and multimedia graphics is inside this book.

BRET TIMMINS is the co-owner of NewSpeak Software in Draper, UT. He has worked on such popular programs as the PC version of NBA Jam and Sega's Dragon's Fire, as well as a music/sound FX driver for Nintendo programming. NewSpeak is currently developing FrameCraft, a Windows-based graphics application. Bret has been writing magazine articles on computer programming since the Commodore 128 was state-of-the-art.

The CD-ROM includes Knob—the author's sample program, a high-speed parallax-scrolling platform game that is used in the book to explore the inner workings of DirectDraw; several custom editing tools including a character editor and sprite converter created at NewSpeak; several handy commercial animation development tools, demo versions of the hottest new Windows 95 games, and the entire source code from the book along with modifications to show even more ways to use DirectDraw.

Basic DirectDraw Topics Include:

- Objects—structure, resolution, color depth, and access
- Surfaces—primary surfaces, creating a DirectDrawSurface with a backbuffer, attaching, and allocation memory
- Color—creating a palette, dealing with YUV, color models, the GDI with DirectDraw palettes
- Blitting—animation using primary, backbuffer, and offscreen surfaces, the flip function, transparent blits, setting color key ranges, animation timing

Advanced DirectDraw Topics Include:

- Alpha Blending—creating an alpha blending surface, using color models
- DirectDraw in a Window—as a GDI strategy, resolving palette conflicts
- Sprite Classes—implementing, data declarations, member functions, exercising and adding to the sprite class, frame animation, movement, rotation, and stretching
- Tiling Classes—advantages of tiling under DirectDraw, smooth scrolling, combining sprites with the tile class, full parallax scrolling, adding to the tile class

Level

Intermediate/Advanced

Games Programming

PC/Windows

Cover art © Bill Westheimer
Cover design by Gary Szczecina



M&T
BOOKS

ISBN 1-55851-460-0



9 781558 514607

US \$39.95

9 0000 >

